Escuela Politécnica Superior
Universidad Carlos III de Madrid

PROYECTO FIN DE CARRERA

Ingeniería Técnica de Telecomunicación
especialidad Telemática

EVOLUCIÓN DE UNA PLATAFORMA DE JUEGOS BASADA EN EJECUCIÓN DE GUIONES

Autor: María Jiménez Trigueros

Tutor: Julio Villena Román

Diciembre 2009

PROYECTO FIN DE CARRERA

Universidad Carlos III de Madrid

Ingeniería Técnica de Telecomunicación

especialidad Telemática

Título: Evolución de una plataforma de juegos basada en ejecución de guiones.

Autor: María Jiménez Trigueros.

Tutor: Julio Villena Román.

EL TRIBUNAL

Presidente: Mª Carmen Fernández Panadero

Secretario: Isaías Martínez Yelmo

Vocal: Jaime de Uriarte de Antonio

Realizado el acto de defensa y lectura del Proyecto Fin de Carrera el día 14 de Diciembre de 2009 en Leganés, en la Escuela Politécnica Superior de la Universidad Carlos III de Madrid, acuerda otorgarle la CALIFICACIÓN de:

Fdo: Presidente Fdo: Secretario Fdo: Vocal

Resumen

Inicialmente, el proyecto era una plataforma de juegos basada en guiones, en la cuál sólo se podía jugar.

En el juego podían aparecer objetos inertes, ascensores que iban de una planta superior a otra inferior (o viceversa) y personajes que podían tener diálogos.

Los personajes que aparecían en la escena carecían de movimiento, o en su defecto el único movimiento que tenían era un pequeño giro en su eje Y. Tras la evolución del proyecto se ha pretendido dar movimiento a los personajes, los cuales podrán tener un recorrido en línea recta paralelo al eje X o Z. Para llevar a cabo la animación del movimiento anteriormente se tienen que haber obtenido unos fotogramas que representen dicho movimiento.

Los diálogos que tenían los personajes eran mostrados por la aplicación en ventanas emergentes, por lo que se pretende incrustar estos diálogos dentro de la aplicación.

El diálogo de un personaje está formado por intervenciones, las cuales podían ser de tipo texto, opción, test, tema y final. Una intervención de tipo texto como su propio nombre indica, sólo contiene texto y sólo tiene un descendiente. Una intervención de tipo opción da a elegir al protagonista una opción para seguir con la conversación, tiene tantos descendientes como opciones posea. Una intervención de tipo tema, es similar a la de tipo texto pero obtiene el contenido de un fichero externo que contiene todos los temas de la aplicación donde será referenciado el contenido por un identificador, tiene un descendiente. Una intervención de tipo test es la forma que tiene la aplicación de evaluar al usuario de los conocimientos expuestos en las intervenciones tipo tema, tiene dos descendientes que definen la evaluación del usuario (aprobado o suspenso). Una intervención de tipo final indica que se ha llegado al final de la conversación.

Con la evolución de la aplicación se ha pretendido añadir otras intervenciones: tipo común y parte común, donde una parte de la conversación pretende ser compartida por varias bifurcaciones. Una intervención de tipo común indica que el resto de la conversación es una parte común por lo que será enlazada con dicha parte común. Por lo tanto esta intervención tendrá un descendiente. Una intervención parte común indica el resto de conversación (intervención/es) que serán compartidas por varias bifurcaciones, es con lo que enlaza la intervención anterior. Tiene un descendiente.

Las intervenciones de tipo test contienen una serie de preguntas para evaluar al usuario. Anteriormente estas preguntas sólo podían ser de tipo test y todas las preguntas de un test debían exponer el mismo número de opciones de respuesta. Con la evolución del proyecto se pretende que esto no sea así y pueda soportar preguntas con distinto número de opciones. A la vez, también se añade la posibilidad de que el usuario pueda introducir la respuesta por teclado (algo que sólo debería ser proporcionado para preguntas de solución única).

En dicho juego no cabía la posibilidad de sonido, por lo que a través de este proyecto se ha intentado dar sonido al juego, tanto ambiente como en el diálogo de los personajes.

Todos los ficheros utilizados por la aplicación (docentes y guión) son documentos XML. Anteriormente debían ser generados introduciendo los elementos soportados por los DTDs correspondientes. Otra evolución del proyecto ha sido tener la posibilidad de generar los documentos docentes (temas y tests) a través de una aplicación donde sólo es necesario introducir los datos docentes.

Lo mismo pasa con el guión de juego, puede ser generado a través de colocar los objetos en el escenario, dar diálogo a los personajes y enlazar fases (si fuera necesario).

Por lo tanto no es necesario de escribir los ficheros XML (con los elementos) puesto que indicando los datos oportunos los ficheros serán creados por la aplicación.

Además, el usuario tiene la posibilidad de guardar una partida y posteriormente reanudarla por donde la había dejado, cosa que antes no era posible.

ÍNDICE

1.- Introducción

1.1. - Marco de trabajo

El panorama de hoy en día en las Tecnologías de la Información y Comunicaciones (TIC) es fruto de los múltiples avances en el ámbito tecnológico. Estos progresos se han traducido, entre otras cosas, en un incremento de la potencia de los ordenadores, posibilitando así la creación de aplicaciones cada vez más complejas y elaboradas.

Actualmente la percepción de un juego como sistema de aprendizaje se encuentra bastante degradada. Debido a esto se opta por Tutores Inteligentes (Intelligent Tutoring Systems, ITS) para la realización de este tipo de aplicaciones, caracterizadas por ser aburridas y no captar la atención del usuario final. Esto hace que toda iniciativa actual creada para el desarrollo de aplicaciones docentes derive en un sistema con escaso atractivo para el sector de interés. Esto tirará por tierra todo el esfuerzo realizado para conseguir tal producto, puesto que en la mayoría de los casos se llega a un programa cuyos usuarios utilizan de forzadamente y al que dedican únicamente las horas que se le hayan impuesto.

La solución de este problema pasa por aunar las características de aplicaciones lúdicas y docentes en un producto final, capaz de ofrecer al alumno entretenimiento a medida que va asimilando contenidos didácticos sin ningún esfuerzo. Dicha aplicación será lo que se puede denominar “Juego aplicado a la enseñanza”.

1.2. – Premisas de la plataforma inicial

Para llegar a la plataforma de la primera versión, se construyó un motor de juegos capaz de dotar al desarrollador de una herramienta potente y sencilla que posibilitaba la creación de juegos complejos con el menor grado de dificultad posible.

Por otro lado, la plataforma realizaba una distinción entre los roles inmersos en la creación de los juegos, pudiendo separar el papel del diseñador y del tutor, encargado este último de llevar a cabo exclusivamente actividades de carácter docente.

Esta división en el desarrollo permite separar la labor de ambos personajes, de modo que el papel del tutor queda apartado del desarrollo del juego, evitándole toda labor de diseño del mismo. En cuanto al desarrollador, es el encargado de la creación del guión que caracterizará la aplicación, siendo éste el único que requiere de unos conocimientos más elevados en lo que respecta a la plataforma.

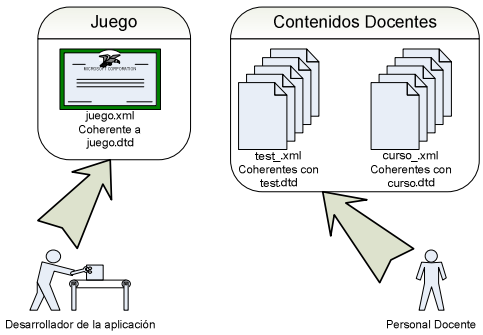

Figura 1: Dualidad en el desarrollo

Para poder llegar hasta esta herramienta fue necesario realizar un estudio comparativo de las tecnologías vigentes hoy en día en lo que a motores de juego se refiere, ya que la elección de dicho motor fue lo que determinó las demás tecnologías utilizadas en el desarrollo de la plataforma. Este proceso comparativo ayudó a decantarse por una tecnología caracterizada por su potencia y versatilidad a la hora de crear nuevos juegos, posibilitando la construcción de aplicaciones dispares entre sí.

Una vez que se eligieron las tecnologías infraestructurales, el trabajo se centró en el desarrollo del nivel intermedio que hizo posible la transformación de una entrada en forma de guión, que caracterizaba una aventura concreta, en dicha aventura en forma de aplicación.

La premisa que guía el desarrollo de la plataforma tiene que ver con la creación de un sistema que posibilite el desarrollo de juegos con fines didácticos, que se ciñan a la idea "aprendizaje basado en juegos". Es decir, aventuras en las que el protagonista aprende un tema en concreto, a medida que avanza en ella. Por ello la estructura de la plataforma resultante se considera óptima para el desarrollo de este tipo de juegos, ya que implementa todas las herramientas necesarias para desarrollar cualquier tipo de interacción de carácter docente tales como generadores de test, expositor de lecciones, etc.

Pero a pesar de esta leve especialización, se pretende que la plataforma también cuente con las herramientas necesarias para el desarrollo de cualquier tipo de aventura que el desarrollador tenga en mente.

Se puede decir que la plataforma que se diseñó no es sino un compilador e intérprete de guiones que definen un juego. Se la puede dotar del nombre 'compilador' puesto que verdaderamente lleva a cabo trabajos de compilación del código fuente diseñado por el desarrollado. Dicho proceso lleva a cabo una validación de la sintaxis del documento así como de todo su contenido, e indica los errores encontrados en él. En cuanto al apelativo de 'intérprete', es obvio que la misión final de la plataforma reside en la lectura e interpretación de guiones, creando la aplicación correspondiente.

Esta aplicación carece de sonido, de movilidad animada de los personajes, aparecen ventanas emergentes para los diálogos de los personajes, temas y/o tests…

1.3. - Objetivos

El objetivo que se pretende conseguir tras la consecución de este proyecto, es la mejora de una plataforma dada para el desarrollo de juegos basada en una ejecución de guiones creada en un PFC anterior.

Como se ha comentado en el apartado anterior, se parte de una plataforma basada en guiones, en la que tras un documento XML bien formado siguiendo una DTD apropiada se genera un escenario de juego en el que el usuario puede recorrer el mundo e interactuar con los personajes.

Una vez que el usuario se encuentra con un personaje, si éste tiene diálogo lo mostrará, pudiendo ser dicho diálogo una intervención, un tema o un test.

Las ventanas de diálogo de los personajes son ventanas emergentes por lo que se pretende incrustar dichas ventanas dentro de la aplicación.

Dichos personajes, aparecían en el mundo sin moverse, o en su defecto realizaban un pequeño giro sobre su eje Y. Tras la realización de este proyecto, lo que se quiere hacer es dar mayor movimiento a los personajes, por lo que podrán hacer un recorrido.

Otro de los objetivos que tiene este proyecto es aportar sonido al juego, así se podrá reproducir tanto un sonido ambiente como un sonido por cada intervención de diálogo.

Dentro de los tests que soportaba la primera versión aparecían unas restricciones: todas las preguntas eran de tipo test, y dentro de un test todas las preguntas tenían que tener el mismo número de opciones. Lo que se pretende conseguir esta vez es que un test soporte tanto preguntas tipo test como respuestas escritas por el usuario, a la vez de poder soportar preguntas test con distinto número de opciones. Así también dentro de un "test" podrá tener estos dos tipos de preguntas (con opciones o respuestas escritas por el usuario).

Otra de las mejoras que se intentará dar será ampliar los tipos de intervenciones. Así se podrá contemplar la posibilidad de que varias opciones tengan una parte en común, por lo que ante una bifurcación se podrá llegar al mismo punto del camino.

Una de las mejoras más importantes que se va a tener en cuenta será la posibilidad de editar dentro de la aplicación todos los documentos XML que usará el juego. Por lo que estará la posibilidad de editar los documentos docentes, tanto temas como tests, así como los guiones de juego. Por lo tanto, el desarrollador no tendrá la necesidad de crear el guión XML del juego

generando todos los elementos necesarios que recoge el DTD correspondiente, puesto que la aplicación los generará por él tras la colocación de los elementos dentro del escenario. Por lo que tampoco el tutor tendrá la necesidad de generar los documentos XML docentes, puesto que tras meter dichos datos en la aplicación se generarán correctamente los documentos correspondientes.

1.4. - Estructura del documento

La memoria se encuentra estructurada en los siguientes capítulos y anexos [14]:

- *Introducción*: Presente apartado en el que se introduce documento.
- *Estado del Arte*: En este capítulo se incluyen puntos que van desde el concepto de los hoy denominados "Juegos aplicados a la enseñaza", pasando por la explicación del funcionamiento de un motor de juegos y una breve descripción de las tecnologías que se miraron en el primer proyecto. En este punto también se llevará a cabo la explicación de las tecnologías empleadas en el desarrollo de la plataforma, describiendo el funcionamiento y estructura de cada una de ellas (Java, JDOM, jPCT y XML).
- *Análisis*: En este punto se explicará cómo se encontraba la aplicación antes de hacer la evolución y cómo quedará, explicando cómo funciona.
- *Implementación de la plataforma*: Expondrá los detalles de más bajo nivel, que van desde la especificación de la estructura y composición de cada uno de los módulos que constituyen la aplicación, hasta llegar al análisis del lenguaje empleado para la composición de juegos con este sistema.
- *Validación de la plataforma*: Capítulo en el que se lleva a cabo una serie de pruebas que aseguren el funcionamiento correcto de la aplicación creada, así como el grado de aceptación de la misma. En este punto se expondrán los resultados a tales pruebas así como el juego de demostración creado para la evaluación de la plataforma.
- *Conclusiones y trabajos futuros*: Capítulo en el que se exponen las conclusiones obtenidas tras la finalización del proyecto, así como las líneas de trabajo futuro planteadas después de la realización del mismo.
- *Anexo I: 'DTD's utilizadas por la plataforma'*: Exposición de las DTD que validan el contenido de los documentos XML que definen el juego creado.
- *Anexo II: 'Manual del desarrollador'*: Mini-manual dirigido al desarrollador de juegos que explica los pasos por los que tiene que pasar en la definición de un guión.
- *Anexo III: 'Manual de utilización (Juegos diseñados para la plataforma)'*: Apartado en el que se exponen las pautas comunes, referentes a la utilización del usuario, de todos los juegos sintetizados por la aplicación.
- *Anexo IV: 'Documentación de las clases Java'*: Apartado en el que se expondrá el contenido de la documentación del programa, generada por Java (javadoc).

2.- Estado del arte

2.1. - Juegos aplicados a la enseñanza

Los avances realizados en el sector de la informática han hecho posible los numerosos intentos que se están propiciando con el fin de crear sistemas capaces de suplir en alguna de sus labores al personal docente, sistemas que se encuentran basados en animaciones simples, simulaciones, entrenamiento asistido, aprendizaje con soporte Web, agentes pedagógicos e ITS ("Intelligent Tutoring Systems" o tutores inteligentes). Dichos sistemas, hoy en día todavía no han llegado a ser totalmente autosuficientes, aunque constituyen herramientas que otorgan un valor añadido al contenido de una materia determinada [9].

El panorama actual se caracteriza por una escasa aceptación de los juegos como sistemas de aprendizaje, por lo que dichos sistemas se basan fundamentalmente en tutores inteligentes (ITS), opcionalmente mejorados por una *realimentación* y un *agente pedagógico*. La *realimentación* proporciona al sistema un conocimiento añadido del progreso del estudiante en su travesía por dicha aplicación, de modo que ésta se pueda adaptar a las necesidades del alumno. Así, por ejemplo, dependiendo del nivel del usuario, la aplicación se comportará de una u otra forma (explicando los contenidos con diferente profundidad, formulando cuestiones de diferente nivel, y cualquier otra decisión que sea considerada). En cuanto al *agente pedagógico,* éste hará las veces de guía, ofreciendo una ayuda contextualizada en todo momento.

Este panorama hace que la mayoría de los esfuerzos se focalicen en sistemas caracterizados por aplicaciones poco atractivas para el sector de interés al que van dirigido, debido fundamentalmente a que todos ellos se caracterizan por una interfaz algo tediosa a la hora de interactuar con el usuario. Dicho desinterés hace que, aunque el esfuerzo para obtener sistemas de una elevada calidad que cubran todas las necesidades docentes de un alumno sea colosal, tales aplicaciones sean inservibles en el peor de los casos debido a que el estudiante únicamente las utiliza de forma obligada.

Viendo esto, se puede considerar que se está ante una situación conflictiva de fácil solución. Esto es debido a que las carencias que demandan los actuales sistemas desarrollados, pueden ser solventadas por las características que poseen los juegos basados en entornos virtuales en los que el protagonista se involucra en un único cometido: finalizar la aventura en la que se encuentra envuelto. Dichos juegos demuestran que tareas costosas (como pasar a la pantalla final de un determinado juego), pueden llegar a ser motivantes a la vez que divertidas cuando forman parte de un contexto atractivo. De este modo, si se consiguiese que el contenido docente que caracteriza los sistemas anteriores estuviese integrado a lo largo de una buena historia, de una forma difuminada constituyendo así una aplicación didáctica atractiva, se llegaría a lo que llamamos "Juegos aplicados a la enseñanza".

Como se puede ver, una solución para crear un material que se adecue a las características demandadas por alumnos y profesores, de modo que se llegue a aplicaciones docentes atractivas para el público al que van destinado, pasa por la combinación de las prestaciones aportadas por juegos basados en entornos virtuales y sistemas de educación basados en un tutor inteligente. Por ello en los siguientes apartados se estudiarán ambas opciones de modo que se puedan obtener las características principales que se demandan de cada uno de ellos para conformar lo que se ha denominado, un juego aplicado a la enseñanza [5].

2.1.1. - Sistemas de enseñanza

Todo sistema de enseñanza se encuentra caracterizado por un ciclo de vida que caracteriza su comportamiento en cada momento. Dicho ciclo de vida se encuentra reflejado en la siguiente figura.

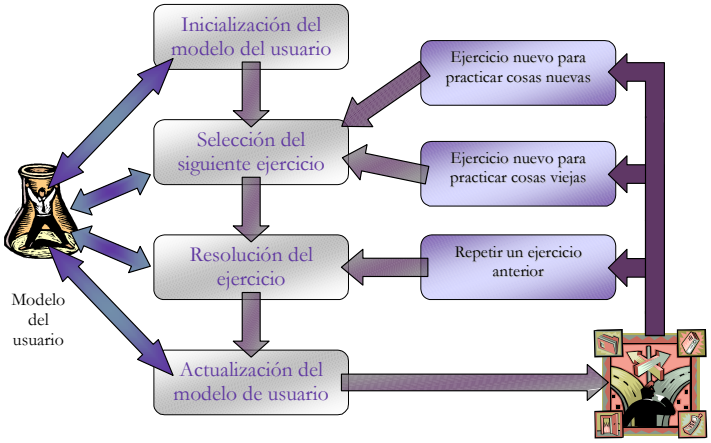

Figura 2. Ciclo de un sistema de enseñanza

A continuación y como ya se ha dicho antes, se procederá a estudiar las características principales de todo sistema de enseñanza:

a. *Modelado del Usuario*

Los sistemas de enseñanza se sirven de un modelado de usuario que representa su progreso a lo largo del juego; de este modo, el comportamiento del programa se adecua a dicho modelo.

Este modelado sería preciso a la hora de implementar las aplicaciones que se persiguen. Dichos datos permitirían al juego saber el estado en el que se encuentra el alumno, pudiendo modificar su comportamiento ante las diferentes situaciones que se pueden dar, como ya se explicó en el caso del tutor inteligente.

De este modo, ante un alumno que no ha resuelto correctamente una prueba, o bien la ha resuelto con bastantes dificultades, el juego respondería pasando a una fase o nivel que perfeccionase las carencias que ha demostrado (así pues las carencias pueden ser diferentes y el juego se adaptaría para completar estas y así no aburrir al alumno con materia que tiene correctamente asimilada).

b. Agente Pedagógico

Personaje o avatar que comparte el entorno de aprendizaje en el que se encuentra inmerso el alumno, de modo que le ayuda a resolver las tareas encomendadas. Este agente detecta el momento más adecuado para aportar una explicación contextualizada que maximiza las posibilidades de asimilación y comprensión de un concepto por parte del alumno.

Si se extrapola esta idea al caso actual, se estará hablando de un personaje que acompañe al protagonista durante toda su cruzada, de modo que le sea de ayuda proporcionándole consejos que le permitan proseguir su aventura (evitando que se quede atascado en alguna fase) en el momento que lo necesite. Esto infunde una confianza extra en el actor principal, potenciando su interés por el juego, debido a que nunca llegará a caer en la desidia producida por la imposibilidad de avanzar en su camino.

c. Gestión de Avance

Cuando se habla de gestión del avance se hace referencia a la posibilidad que todo sistema de enseñanza ofrece a su usuario de poder repetir ejercicios o lecciones anteriores con el fin de optimizar sus conocimientos en dicha materia. Como se puede observar en el ciclo de un sistema de enseñanza, esta característica forma parte de su funcionamiento (ver figura 2).

Sería preciso extrapolar esta idea al caso de los juegos aplicados a la enseñanza, de modo que la aplicación permitiese retroceder a escenas pasadas para poder recordar, repasar o perfeccionar dicho nivel, con su consecuente modificación en el modelado del usuario actual.

d. Modulo Pedagógico

Compuesto por todos los problemas y ejercicios que se podrán formular al protagonista dado su estado de avance en el juego. Dicho módulo vendrá condicionado además por el modelado de usuario, de tal forma que dependiendo de las características impresas por dicho usuario durante el transcurso de la historia en este modelado, el módulo estará formado o no por determinadas preguntas.

Es preciso incorporar esta característica al sistema a desarrollar, debido a que todo procedimiento que pretende inculcar una enseñanza (sea ésta la que sea), necesita de una herramienta evaluadora que pondere el baremo de conocimientos asimilado por el alumno, de modo que pueda adaptar su comportamiento a tal realimentación.

2.1.2. - Videojuego basado en un entorno virtual

Todo juego se encuentra caracterizado por un ciclo de vida que detalla su comportamiento en cada momento. Dicho ciclo de vida se encuentra reflejado en la siguiente figura.

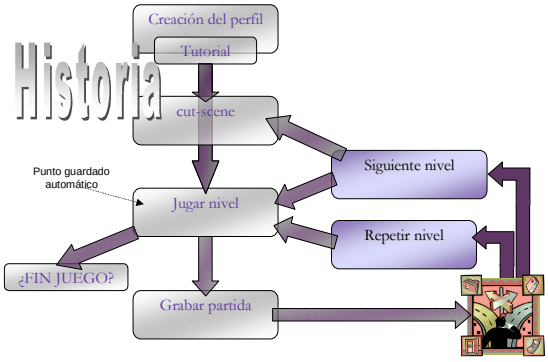

Figura 3: Ciclo de un videojuego

La característica fundamental que aportan los videojuegos a los sistemas de enseñanza basados en juegos educativos se encuentra resumida en la frase de Malcolm Gladwell, prestigioso periodista del "The Washington Post", cuando se refirió a la simpleza del éxito de un famoso programa educativo de televisión: "si puedes mantener la atención de los niños, puedes educarles" [3].

El sistema a desarrollar ha de tener la capacidad de atraer al usuario por sí mismo, siendo capaz de postrarlo durante horas delante de él. Sólo ésto permitirá que el protagonista dedique la cantidad de horas necesarias haciendo uso de la aplicación diseñada, lo que le permitirá llegar a asimilar todo el conocimiento que se le pretende inculcar. A su vez, todo esto inducirá al usuario a entrar en una espiral positiva, en la que dicha motivación hará incrementar su capacidad de asimilación, lo que se traducirá en un aumento del tiempo pasado inmerso en el juego.

La motivación y capacidad de atracción de la que se habla forma parte de la esencia de todo videojuego caracterizado por una buena historia de trasfondo. Lo que se pretende en el aprendizaje basado en juegos es acercar dicha esencia a los sistemas de aprendizaje actuales, de modo que la utilización de estos deje de ser una actividad agobiante caracterizada por la desidia de los usuarios, para pasar a convertirse en una aplicación interesante capaz de atraer por sí sola al actor. De este modo dicha utilidad se tornará en una tarea motivante capaz de envolver al usuario (cuyas actuaciones se orientarán a descubrir hasta dónde le conduce el juego) de un entorno educativo del que irá asimilando, con muy poco esfuerzos, los conocimientos necesarios acerca de un tema en concreto.

Otra de las características que se pretende importar de los videojuegos, y que es sumamente imprescindible para el éxito de los sistemas bajo estudio, radica en la historia de trasfondo utilizada como hilo conductor a lo largo de un juego. Dicha historia proporciona una idea de unidad en los avatares que componen la aventura.

Es fundamental que el sistema de aprendizaje que se pretende diseñar cuente con dicha conectividad entre sus elementos, pasando de ser una mera sucesión de ejercicios independientes, a una aplicación coherente en su contenido, aplicación por la que el protagonista pueda navegar sabiendo que ha partido desde un inicio determinado, con la finalidad de arribar a un final existente, únicamente siguiendo el guión que él determine. Esto hará que el sistema de enseñanza quede inmerso en la infraestructura de la aplicación, no resultando explicito así al usuario y ocultando así la parte menos atractiva y exasperante.

Analizando lo expuesto hasta ahora, se puede observar cómo la aplicación que se busca tiene sus fundamentos en un sistema de aprendizaje cuya interfaz estará implementada en un videojuego resuelto en un entorno virtual. Para completar con éxito dicho proyecto será necesario aunar ciertas propiedades que caracterizan tanto a los videojuegos como a los sistemas de aprendizaje actuales (ITS, agentes pedagógicos, simuladores, etc.) por separado, llegando a un producto final atractivo a la vez que didáctico. La aplicación de enseñanza resultante contará con los siguientes avatares:

- *Inicio de la utilidad*: Creación del modelo que caracterizará al usuario durante todo el transcurso de su aventura, y que se irá modificando a medida que ésta avance.
- *Tutorial del juego*: Fases iniciales del juego caracterizadas por rutinas en las que se pretende evaluar el conocimiento previo del usuario en lo que respecta a la materia que se va a impartir (creando así un modelado de usuario inicial válido), además de familiarizar a este último con el entorno por el que se desenvolverá.
- *Avance de nivel*: Equivale a resolver con éxito algún tipo de prueba o ejercicio planteado por el módulo pedagógico.
- *Almacenar los progresos realizados*: Supone un almacenamiento del estado en el que se encuentra la historia además del modelado de usuario en dicho momento.
- *Recuperar los progresos realizados*: Proceso contrario al explicado en el punto anterior. En este caso se cargarán tanto el avance como el modelo del usuario que se almacenó en un determinado momento, retornando así la historia al punto donde se dejó.
- *Finalizar la aventura*: Supone haber llegado a un nodo final, ya sea intermedio o absoluto, del grafo que caracteriza la aventura, reconociéndose así la correcta asimilación de los conocimientos necesarios en dicho punto.
- *Fracaso en la superación de un nivel*: Evento producido cuando el usuario ha sido incapaz de superar las pruebas formuladas en un cierto instante por el modulo pedagógico. En tal caso, el sistema proporcionará los medios que vea necesario para inculcar los conocimientos de los que carezca el usuario en dicho instante (pasándolo a un nivel paralelo específico, tratando de solventar la carencia puntual en sus conocimientos, o cualquier otra solución que el sistema considere necesaria y tenga implementada).

2.2. - Motores gráficos

2.2.1. - Conceptos generales

Todo motor gráfico implementa la parte encargada de la gestión y actualización de los gráficos 3D en tiempo real de todo software cuya interfaz se encuentra implementada por un entorno virtual, convirtiéndose así en uno de los componentes más importantes a la hora de desarrollar juegos 3D.

Debido a su complejidad, determinadas empresas se encargan de desarrollar dicho software a la vez que sus correspondientes API's, que posibilitan al usuario hacer uso de sus prestaciones de una forma ordenada y sencilla. Debido a esto, existen diversas tecnologías en

cuanto a motores de juego se refiere, por las que cada usuario inmerso en la creación de un juego basado en un entorno virtual se podrá decantar.

En este punto se pretende abordar los conceptos principales que caracterizan a todo motor gráfico, explicando alguna de las posibles características que pueden llegar a implementar, para finalizar exponiendo los motores gráficos más importantes existentes hoy en día, así como cada una de las características que definen a cada uno de ellos.

Figura 4: Imagen de juego basado en el motor gráfico Fly3D

El funcionamiento interno básico que forma la infraestructura de cualquier motor de juegos está resumido en una idea fundamental: esta tecnología tiene como principal cometido enviar a la aplicación que se sirve de ella los datos que necesita el próximo 'frame' para su representación. En estas palabras queda sintetizado el funcionamiento de todo motor de juegos, aunque el trasfondo de este concepto va a ser sumamente amplio.

Estos datos enviados resultan de la sintetización de multitud de variables que forman parte del funcionamiento del motor y que constituyen la base de cualquier juego. Así por lo tanto, el resultado final devuelto por un motor de juego pasa por haber realizado un procesado previo de la posición actual que se quiere representar, la posibilidad de haberse producido algún tipo de colisión, las posiciones de los elementos en el escenario virtual que los envuelve, y un largo etcétera que construyen cada uno de los detalles del juego.

Se observa cómo las labores realizadas por el motor son labores de bajo nivel en lo que se refiere al tratamiento de todos los objetos que componen el entorno, abstrayendo así al usuario de dicha complejidad y proveyéndole de material de alto nivel que más tarde utilizará en sus aplicaciones (p.ej: la imagen final una vez procesado el movimiento, colisiones, escenario, que se le proveerá al diseñador del juego para su posterior representación en pantalla, o el evento producido por una colisión así como aquellos argumentos que la caractericen, etc.). Esto aporta una facilidad añadida en la creación de estos tipos de juego, debido a que el diseñador no tendrá que preocuparse por tareas de tan bajo nivel, resolviendo éstas a través de la API que ofrece el motor gráfico elegido [6].

Una vez descritos los fundamentos de todo motor de juegos se pasará a explicar algunas de las técnicas que se utilizan a la hora de realizar su labor [7]:

- *Renderizado*: Se denomina así al proceso de cálculo complejo realizado por un ordenador con la finalidad de generar una secuencia de imágenes 3D, tratando las texturas de dicha imagen así como el comportamiento de las luces existentes. De esta forma el procesador entenderá las gráficas de los juegos 3D como coordenadas en un plano cartesiano de tres dimensiones XYZ.

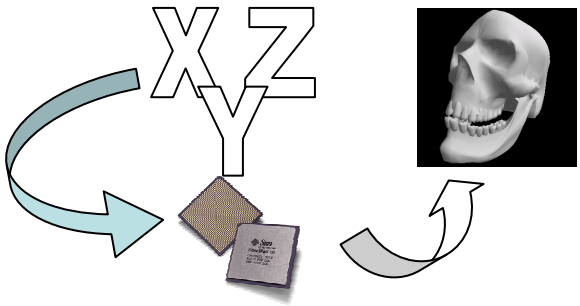

Figura 5: Proceso de renderización

- *Árboles BSP*: Estructura de datos utilizadas por el motor de juegos con el fin de organizar objetos dentro de un espacio. Adicionalmente a esto, estos árboles tendrán aplicaciones en la renderización de áreas ocultas así como en el trazado de rayos.

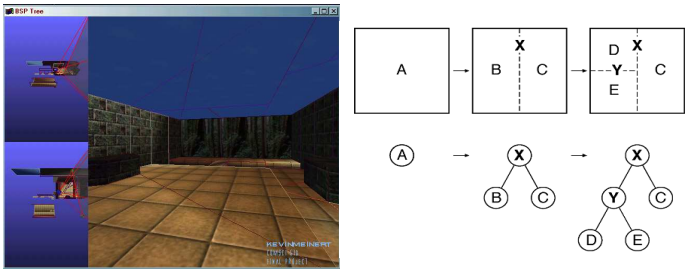

Figura 6: Árbol BSP

- *Texture mapping*: Método por el que una textura es aplicada a la superficie de una figura. Este proceso es semejante a envolver un objeto con un papel de regalo, siendo el papel de regalo la textura en cuestión [13].
- *Radiosidad*: Proceso por el cual el motor de juegos realiza el cálculo de la iluminación global de un ambiente cerrado, considerando todos y cada uno de los focos de luz existentes. Este proceso busca en todo momento el equilibrio entre la energía emitida por los focos de luz y aquella que es absorbida por los objetos existentes en el entorno actual.

Figura 7: Ejemplo del tratamiento de Radiosidad

- *Mip-Mapping*: Técnica por la que el motor gráfico realiza un manejo de texturas de un polígono tal que permita dotar de mayor calidad y nivel de resolución a aquellos objetos más cercanos al protagonista, consiguiéndose así un mayor rendimiento. Esta técnica atiende a parámetros tales como el ángulo de vista del jugador así como las condiciones del juego en el que se desenvuelve.

Figura 8: Ejemplo de efecto mip-mapping

- *Bump-Mapping*: Técnica por medio de la que se puede agregar un mayor detalle a una imagen sin necesidad de aumentar el número de polígonos. Este proceso está basado en algoritmos que parten de una textura inicial convirtiéndola en otra de modo que inserta pequeños 'Bump mapping' en la superficie del objeto dotándolo así de unta textura determinada sin cambiar la superficie del objeto.

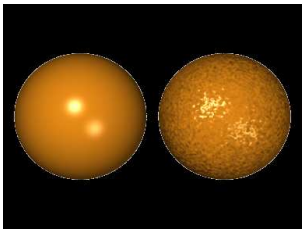

Figura 9: Ejemplo de Bump-Mapping

- *Portal rendering*: Algoritmo que determina la visibilidad en una escena. Un nivel de un juego puede estar constituido por una multitud de polígonos de los cuales solo unos pocos podrían ser visibles por pantalla en un momento determinado. Este algoritmo permitirá decidir entre cuales serán visibles y cuales no [13].
- *Lightmaps:* Técnica por medio de la que se pueden llegar a combinar dos texturas llegando a un efecto final resultado de la unión de ambas. Esta técnica permite resolver efectos como el de la radiosidad con sólo la utilización de una segunda textura que haga los efectos correspondientes a la degradación de luz.

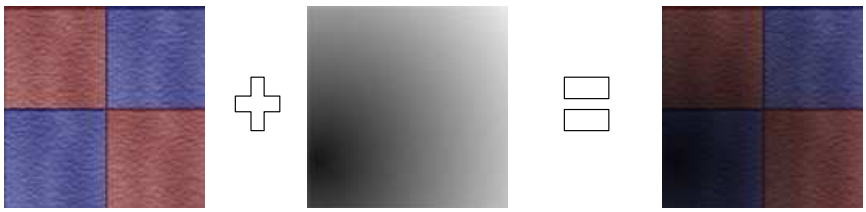

Figura 10: Efecto Lightmaps

Existen numerosos tipos de técnicas además de las mencionadas hasta este punto, pero las enumeradas aquí son las más comunes dentro de los motores de juegos. Aún así, como se ha dicho, hay infinidad de técnicas propias de determinadas tecnologías, como ocurre con el caso de la detección de colisiones.

2.2.2. - Comparativa de motores gráficos

En la implementación de la primera versión de la plataforma se analizaron varios motores gráficos para poder decidir cuál de ellos sería válido para dicha aplicación.

Los siguientes motores (con una muy breve descripción) son los que se analizaron [4]:

- *3D GameStudio*: Kit de desarrollo de juegos por ordenador que consta de un motor 3D así como otro para gráficos 2D, además de un editor de niveles y modelos que construyen por completo el entorno virtual. Adicionalmente a los componentes enunciados hasta ahora, el motor cuenta con un compilador y depurador de scripts del que se sirve a la hora de crear los juegos, además de múltiples librerías de complementos útiles para la creación de aventuras gráficas, como van a ser modelos, texturas, entornos, y demás componentes.

  No requiera un elevado conocimiento de programación para su utilización.

- *Crystal Space*: Software libre (LGPL) implementado en C++ y caracterizado por su gran portabilidad y escalabilidad proporcionada por el sistema de plugins.

  Requiere un conocimiento elevado en lenguajes de programación de alto nivel para poder hacer uso de sus prestaciones, a las cuales se accederá por medio de API's definidos para tal cometido.

- *Genesis3D*: Motor de juego que implementa las herramientas necesarias para la visualización de escenas tridimensionales en tiempo real, posibilitando así el diseño de aplicaciones gráficas 3D de altas prestaciones.
- *Nebula*: Motor de juegos de calidad profesional y de libre distribución. Se encuentra desarrollado en C++ y orientado a objetos, de modo que sus clases se cargan de forma independiente durante el tiempo de ejecución. Entre otros, los objetivos que busca satisfacer esta tecnología se encuentran ligado a la independencia de plataformas.
- *Ogre*: Motor gráfico escrito en C++ caracterizado por su flexibilidad. Su funcionamiento se encuentra orientado a escenas, siendo su principal cometido el dotar a los diseñadores de una herramienta sencilla e intuitiva a la hora de realizar aplicaciones que utilicen hardware de aceleración 3D.
- *Torque Game Engine*: Motor de juego enfocado al desarrollo de simulación de misiones militares. Dentro de su implementación incluye herramientas para la creación de terrenos, superficies acuáticas, interiores estilo portal y sistemas de partículas.
- *jPCT*: Es el motor utilizado por la aplicación, por lo que en el apartado siguiente está más detallado.

2.3. - Motor jPCT

En la realización del proyecto se optó por el motor de juegos jPCT, y en la evolución de este proyecto no se planteaba cambiar el motor.

Motor gráfico basado en el lenguaje de programación Java, que provee una API pequeña, rápida y sencilla para la renderización de gráficos 3D en applets y aplicaciones.

Este software se ejecuta bajo cualquier máquina virtual de Java que soporte al menos Java 1.1, de modo que según sea mayor la versión instalada puede hacer uso de prestaciones más avanzadas. Así, por lo tanto, si se tiene instalado Java 1.2 se podrá hacer uso de la subclase `BufferedImage`, o si por el contrario se tiene instalado Java 1.4 será posible hacer uso de la aceleración por hardware vía OpenGL, además de todas las prestaciones implementadas en las versiones anteriores. Por ello se recomienda tener una versión igual o posterior a la 1.4 para el correcto funcionamiento de la aplicación que se tiene ideado realizar.

jPCT provee de las herramientas necesarias para la creación de juegos basados en entornos 3D en un corto plazo de tiempo y sin ningún tipo de necesidad de acudir a librerías externas, ya que la API del motor cuenta con todo tipo de complementos necesarios en la creación de aplicaciones, como rutinas que detectan colisiones, herramientas para la creación de la GUI, etc. De este modo, las características principales de este motor de juegos son:

- Características del motor:
  - Modelos cargados a partir de archivos 3DS, MD2, ASC o XML.
  - Soporte para octrees y ‘portal rendering’.
  - Animación basada en keyframes.
  - Soporte para un ilimitado numero de fuentes de luz.
  - Soporte de luz ambiente, difusa y especular.
  - Construcción de modelos a partir de primitivas como conos, cubos, esferas, etc.
  - Mapeado esférico del entorno.
  - Detección de colisiones a través de varios mecanismos que estudiaremos más adelante.
  - Efecto de transparencia

- Billboarding
- Efectos que producen una mejora en el control de la cámara.
- Bump-Mapping.

- Características de la renderización hardware.
  - Soporte de escalado RGB.
  - Soporte de pantalla completa en la ventana de OpenGL.
  - Efectos avanzados de niebla y transparencia.
  - Uso de multitextura.
  - Uso de LWJGL.

- Características de la renderización software.
  - W-Buffering de 32 bits.
  - Uso de Bump Mapping.
  - Soporte de escalado RGB.

Así pues se puede considerar el motor jPCT como un software completo en cuanto a prestaciones para la realización de aplicaciones gráficas. Esto exige una gran potencia de la máquina que se encuentra debajo, requiriendo una velocidad de 500 MHz para aplicaciones muy sencillas, y velocidades mayores según se vaya aumentando la complejidad de la misma.

Esta indiscutible necesidad deriva de la aparición de los gráficos 3D, de modo que cuanto mayor sea la potencia del procesador se contará con mejores prestaciones a la hora de representar los diferentes frames. Así la potencia solicitada a un procesador depende de varios parámetros cómo la complejidad de la aplicación implementada, así como el tipo de renderización (software o hardware).

2.3.1. - Estructura Básica

Este apartado está destinado a reflejar la estructura básica de jPCT, realizando un repaso breve de los conceptos básicos sobre los que se sustenta.

'World' (Mundo)

Es la clase más importante en jPCT, aunque no la más potente, que constituye el entorno donde se carga toda la escena. Una aplicación sin al menos un objeto 'World' no tendría ningún sentido, debido a que no existiría la base necesaria para asentar todos los objetos que forman parte del juego.

En resumen, se puede definir el objeto 'World' como el mundo donde se desenvolverá toda la aventura gráfica. En su creación parte como un ente vacio que habrá que caracterizar y al que habrá que añadir objetos tales como el entorno, los personajes, y todo aquello que queremos que forme parte de la aventura. Debido a esto, un objeto 'World' contiene a su vez, ligados a él, objetos pertenecientes a la clase 'Object3D' que constituyen los objetos que forman parte del entorno y que serán renderizados. Estos objetos se crearán por separado, y una vez inicializado el objeto 'World' se le asociarán cada uno de éstos, de modo que cada uno de los objetos que se desee renderizar tiene que estar añadido al mundo que caracteriza la aplicación.

Un objeto 'World' puede tener asociados otros objetos diferentes a los 'Object3D' mencionados antes, como puede ser el caso de objetos de tipo 'Camera', 'Light' y 'Portal'.

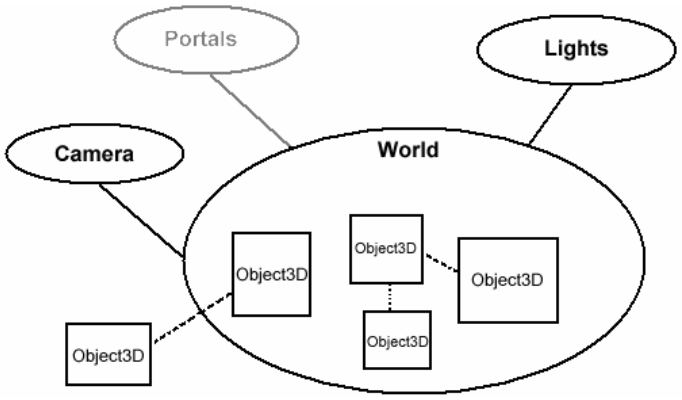

Figura 11: Objeto 'World'

Respecto al papel del objeto 'Camera', todo mundo tiene que tener al menos un objeto de este tipo haciendo las veces de cámara. Esta entidad será la que proporcione información acerca de qué objeto se encuentra a la vista dentro de una escena determinada y por lo tanto será renderizado.

Como se puede ver, todo mundo además tiene que contar con al menos una luz capaz de iluminar la escena que se está representando, para ello se asociará al menos un objeto de la clase 'Light' al objeto 'World' creado. El número de luces asociadas a un mundo en concreto puede variar a gusto del desarrollador, pudiéndose insertar y eliminar de la escena por medio del objeto 'World'.

Además de los objetos 'Camera', 'Light' y 'Object3D' se puede distinguir la presencia de un objeto 'Portal'. Todo mundo propio del entorno jPCT requiere tener un objeto de esta clase, y en el caso de no existir explícitamente, se creará una instancia vacía del mismo.

Por último y para terminar de explicar toda la figura anterior, véase detalladamente la unión existente entre los diferentes objetos de la clase 'Object3D'. Como se puede observar, un objeto puede estar relacionado con varios objetos de la misma clase, siendo esta relación una relación de herencia. Esto es así ya que pueden existir objetos hijos de otro objeto padre, de modo que herede de este último todas las transformaciones que se realicen sobre él.

Esto servirá para implementar los llamados objetos 'dummy', que son simples objetos vacíos que hacen las veces de contenedor de transformaciones. De este modo, y como se ve en la figura, se puede tener un objeto de este tipo no perteneciente al mundo (por lo que no se renderizará), del que hereden varios objetos internos de tal forma que cualquier transformación aplicada a este objeto virtual será común a todos los objetos que hereden de él, generando así un efecto múltiple entre varios objetos que constituyen el mundo con tan solo cambiar una característica del objeto padre.

'Object3D' (Objeto 3D)

Todo aquello que pueda ser renderizado por jPCT se encontrará sintetizado por esta entidad. En la siguiente figura, igual que en el apartado anterior, se muestra un esquema de las posibles relaciones que puede tener un objeto de esta clase. Dentro del esquema se pueden distinguir uniones con líneas continuas que implican uniones obligatorias, y relaciones expresadas con líneas discontinuas cuyo significado implica asociaciones opcionales.

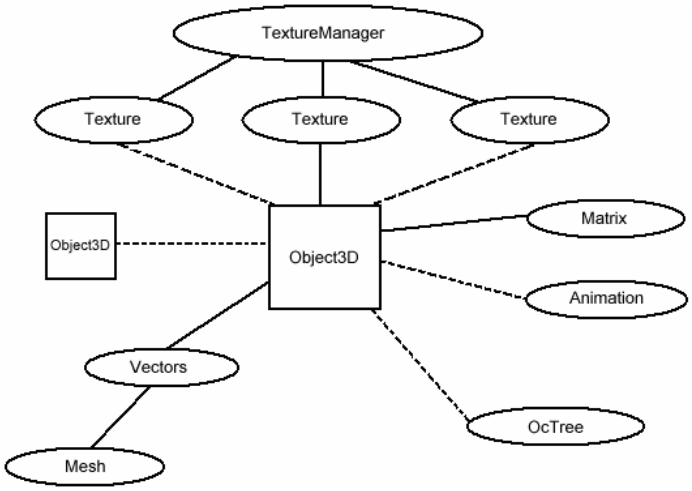

Figura 12: Object3D

Como se puede observar, todo objeto de la entidad 'Object3D' tiene que estar unido con al menos una textura que lo caracterice, un objeto de tipo 'Matriz' y un objeto de tipo 'Vector' que más tarde caracterizaremos. A su vez se puede ver que opcionalmente puede estar relacionado con un objeto 'Animation' que posibilite implementar una animación del elemento en cuestión, un objeto 'OcTree', y como se dijo anteriormente, otro objeto de la misma entidad que haga las veces de padre del que se hereden todas las trasformaciones.

Los vectores implementados por la clase 'Vector' no son interesantes desde el punto de vista del desarrollador, ya que se encuentran casi totalmente ocultos al usuario de la API.

En cuanto a las texturas, todo objeto de esta clase tiene que pertenecer a uno y solo un manejador, implementado por un objeto de la clase 'TextureManager' que caracterizará a uno o varios objetos.

Es importante volver a mencionar la necesidad que tiene un objeto de estar relacionado con al menos una textura. De esta forma, todo objeto inicialmente se caracteriza por contar con una textura totalmente blanca, que seguirá vigente mientras que no se le aplique otra nueva.

Por otro lado, el objeto perteneciente a la clase 'Matrix' será el encargado de realizar cualquier tipo de transformación sobre los objetos, tales como rotaciones, translaciones y escalado.

Adicionalmente a todas las entidades mencionadas hasta el momento, existen otras entidades opcionales características del objeto en cuestión, como pueden ser las entidades de tipo 'Animation', encargadas de realizar cualquier tipo de animaciones basadas en 'keyframes' (superposición de 'frames'), y no en la modificación o transformación de la malla geométrica asociada al objeto tridimensional en cuestión.

'Vector' y 'Mesh' (Vector y Malla)

Los vectores son objetos pertenecientes a la clase 'Vector' que no es pública desde la versión 0.97 del jPCT. Debido a esto, es obvio que dichos objetos no tienen relevancia alguna de cara al desarrollador, ya que son entidades intermedias que sustentan el funcionamiento del motor y que únicamente éste utiliza.

En cuanto a los objetos de la clase 'Mesh' contendrán la información referente a la geometría de uno o varios objetos. Como se ha dicho, una misma malla puede ser compartida por diversos objetos, de modo que una modificación de la misma, se traducirá en cambios visibles en todos los objetos relacionados con ella.

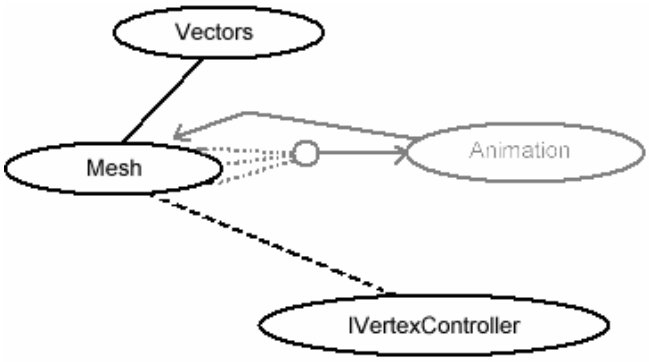

Figura 13: Asociación de Vectores y Mallas

Cuando se habla de modificación de la malla, están implícitas dos posibles iteraciones que cambian el aspecto de la misma. La primera de las iteraciones posibles vendrá derivada por el uso de una animación basada en la sucesión de 'frames'. Dicha animación tiene que ser asignada a un objeto para que tengan algún sentido y, por lo tanto, dicha almacenada en las mallas de los objetos en cuestión, siendo posible compartir varias mallas entre diversos objetos, pudiendo animar varios objetos al mismo tiempo con solo aplicar la animación a uno solo de ellos que comparta esta malla en común.

La segunda forma de manipular una malla es más directa a la vez que complicada. Esta transformación se basa en la implementación de la interfaz 'IVertexControler', la cual proporcionará herramientas que posibilitan cambiar vértices de forma directa en la malla que compone dicho objeto.

'Texture' y 'TextureManager' (Texturas y manejador de texturas)

El manejador de texturas es básicamente un contenedor de texturas, con la peculiaridad de que únicamente puede existir uno en una aplicación. De esta forma se podrá asignar una misma textura a uno o varios objetos si dicha textura se encuentra asociada al manejador utilizada por todos ellos.

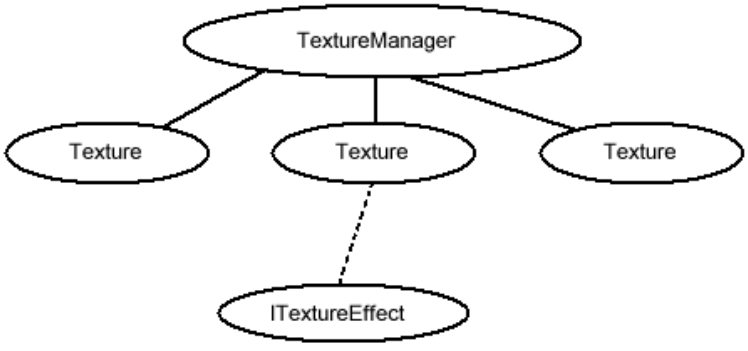

Figura 14: Objetos TextureManager & Texture

En lo referente a las texturas, es preciso destacar la interfaz 'ITextureEffect', cuya implementación puede ser asignada a una entidad de este tipo determinada de modo que posibilite al desarrollador llevar a cabo una manipulación de los píxeles que la componen durante el tiempo de ejecución.

'Matrix' (Matriz)

Herramienta utilizada por el motor de juegos para llevar a cabo rotaciones y translaciones de objetos. La clase Matrix provee al desarrollador de métodos útiles en la manipulación de estas matrices 4x4.

'FrameBuffer'

Herramienta que utiliza el motor gráfico para llevar a cabo la renderización de escenas. jPCT distingue entre renderización y conversión de la escena renderizada, siendo este último el proceso por el cual se dibuja la escena correspondiente dentro del buffer ('FrameBuffer').

Volviendo a retomar lo que ya se dijo antes, es preciso mencionar la posibilidad del motor que permite llevar a cabo un cambio de renderización durante la ejecución de una aplicación concreta. De este modo, si se quisiera alternar a una renderización por hardware habría que eliminar la renderización software para añadir la primera, y ocurre lo mismo con el caso contrario. Pero existe la posibilidad de no tener que eliminar el tipo de renderización existente, lo cual no implica que queden ambas activadas, desactivándose la que está vigente hasta ahora para dar lugar a una renderización distinta, caracterizada por los nuevos parámetros. El encargado de llevar a cabo dicha conmutación es el objeto 'FrameBuffer'.

En la siguiente imagen, se pueden observar las diferencias existentes entre la renderización hardware (OpenGL) o software en lo que a prestaciones gráficas se refiere. Ambas obtienen resultados similares, aunque en la segunda de ellas se puede apreciar texturas menos suaves y un pixelado considerable. Las diferencias mayores entre ambas posibilidades se pueden apreciar sobre todo en el rendimiento que consiguen cada una de ellas.

Renderización Hardware (OpenGL) **Renderización Software**

Figura 15: Renderización Software VS Hardware

'Config' (Configuración)

Finalmente, resulta obligado mencionar la clase 'Config', que proveerá de innumerables métodos y atributos estáticos que posibiliten realizar un cambio en la configuración y comportamiento del motor jPCT.

Son bastante numerosos los parámetros sobre los que se puede actuar con estas herramientas, entre los que se encuentra el modo de renderización, la caracterización de cada una de estas, optimización del motor para un determinado entorno (outdoor frente a indoor), número límite de polígonos visibles en la renderización de una escena, etc.

Sería preciso acudir a la documentación de esta clase para poder acceder a todas sus posibilidades, debido al amplio rango de parámetros de configuración existentes.

2.3.2. - Formatos 3D soportados

A continuación se enunciarán los formatos de archivos que caractericen a un objeto tridimensional, soportados por el motor jPCT.

3DS

Formato usado por el programa 3DStudio. El motor únicamente se servirá de los modelos poligonales de estos archivos, ignorando cualquier otra información de valor añadido.

Éste procederá a cargar cada uno de los modelos poligonales del archivo especificado, devolviendo un array que contenga cada uno de estos objetos tridimensionales. Adicionalmente, serán cargadas las texturas correspondientes al objeto, definidas en el archivo .3ds, pudiendo así cargar objetos enteramente definidos.

MD2

Este tipo de archivo es el utilizado por Quake II para definir los modelos utilizados. MD2 es capaz de soportar animaciones basadas en 'keyframes', por lo que será la extensión de archivos que se utilizarán para definir las animaciones de los personajes dentro de una aplicación.

En este caso, a la hora de cargar este tipo de archivos, se ignorarán los nombres de las texturas correspondientes al modelo (lo que no se hacía en los archivos .3ds), de modo que será preciso cargar y asignar cada una de éstas al modelo en cuestión.

ASC

Archivos antiguos aunque populares, basados en el formato tipo ASCII utilizado por 3DStudio en sus comienzos. Se caracterizan por su sintaxis simple, que los hace vulnerables a ser soportados por muchos conversores, propiedad que les convierte en un formato bastante atractivo.

JAW

Formato casi desconocido basado en ASCII y utilizado por el motor de juegos JAW 3D. Se caracteriza por su facilidad, lo que le hace propicio a ser objetivo de muchos conversores a igual que ASC.

XML

Formato basado en un lenguaje definido por la DTD que definió jPCT en su momento. Este tipo de archivos permite cargar enteramente una escena desde un único archivo XML.

El encontrarse basado en XML, aporta la sencillez y facilidad de comprensión que caracteriza a todos los archivos de este tipo. La DTD que define el lenguaje utilizado para la definición de cada uno de estos archivos puede consultarse en el anexo I.

2.3.3. - Detección de colisiones

Se ha querido dedicar un apartado entero a la detección de colisiones implementadas por jPCT, ya que ésta es una de las características más importantes de la implementación del motor.

Un motor que no implementa un sistema de detección de colisiones resulta casi enteramente inútil, puesto que no será capaz de detectar eventos producidos por la colisión entre los personajes, paredes, elementos del entorno, etc. que constituirán el escenario donde se desenvuelve la aplicación creada. Esto supondrá no poder actuar frente a dichos acontecimientos.

Es por ello por lo que jPCT define 3 rutinas diferentes que implementan 3 modos distintos a la hora de realizar la detección de colisiones. Dichas modalidades serán las siguientes:

Detección de colisiones basado en 'Rayos'

Modalidad sencilla de comprender, aunque no muy potente. Se basa básicamente en la construcción de un rayo (con la longitud correspondiente al movimiento que se quiere realizar) desde el centro de la entidad (cámara u objeto) que queremos evaluar, en la dirección específica. De este modo se comprobará si dicho rayo intersecta con algún polígono existente en la escena, produciéndose en caso afirmativo una colisión.

Este tipo de detección será empleado para objetos de poca envergadura (balas, rayos láser,…) debido a las limitaciones que presenta su utilización en objetos mayores.

Detección de colisiones basado en 'Esferas'

Modalidad en la que se engloba la entidad (objeto o cámara) en una esfera que cubra todo su cuerpo. De este modo se evaluará si se produjo una colisión entre ésta y cualquier polígono existente en la escena.

Esta rutina es un tanto más lenta que la anterior, pero implica una mayor potencia, proporcionando una aproximación más fiel a la estructura de la entidad y suficientemente buena en ciertos casos.

Para comprobar si se produjo algún tipo de colisión, se realizará una translación de la esfera que engloba al cuerpo de la entidad, de tantas unidades como se indique en el programa. De este modo, se comprueba si en dicho trayecto la esfera ha intersecado con algún polígono de la escena, parando el movimiento del personaje en tal caso y devolviendo los parámetros que describan tal evento.

Detección de colisiones basado en 'Elipsoides'

Algunas veces la abstracción de la entidad realizada mediante una esfera no es suficiente. Son casos en los que es preciso un mayor refinamiento del contorno de la entidad, debido a motivos puramente funcionales. Esto se solventará convirtiendo la esfera anterior en una elipse, que de nuevo envuelva a la entidad en cuestión.

Dicha elipse es más conveniente para la representación de humanos, animales y otros muchos objetos. Por ello, en estos casos es necesario refinar la rutina empleada, de modo que se consiga un mejor funcionamiento de la misma.

Por lo demás, esta modalidad es similar a la detección de colisiones basado en 'Esferas', siguiendo sus mismos principios.

2.3.4. - Técnicas avanzadas utilizadas por jPCT

En este apartado final se van a enunciar ciertas técnicas avanzadas que implementa jPCT, y que lo distinguen de los demás motores gráficos.

Multi-texturing

Siempre y cuando se esté utilizando OpenGL para la renderización de la escena, jPCT permitirá tener múltiples texturas por polígono. En caso de utilizar renderización por software no se podrá contar con este soporte, por lo que simplemente se ignorarán las texturas adicionales, quedándose simplemente con la primera de ellas.

Alpha Channel

De nuevo, si se utiliza OpenGL para llevar a cabo la renderización de las escenas, existe la posibilidad de hacer uso de esta técnica consistente en aplicar una cierta transparencia a una

determinada textura. De este modo, si se aplicase multi-textura combinada con la técnica que ahora nos acontece, se podría llegar a sintetizar efectos semejantes a los mostrados en la siguiente figura.

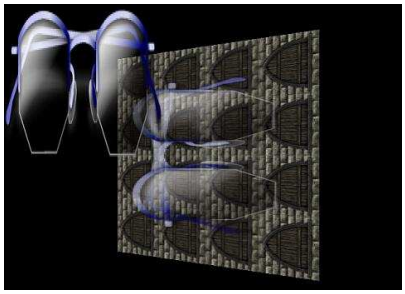

Figura 16: Técnica alpha channel

Aplicaciones con múltiples mundos

Para aplicaciones simples, trabajar con un solo objeto 'World', es decir, un solo mundo, suele ser suficiente. Pero existen aplicaciones que requieren contar con múltiples instancias de objetos 'World', definiendo cada uno de ellos un mundo diferente.

En dichas aplicaciones se tiene que tener en cuenta que un objeto perteneciente a la clase 'Object3D' únicamente puede pertenecer a un mundo determinado, por lo que si se quisiese representar el mismo objeto en diferentes mundos, sería necesario contar con diferentes objetos iguales.

Detección de colisiones avanzada

Por defecto, jPCT realiza una detección de colisiones con la malla que caracteriza al objeto tridimensional en cuestión. Pero esta malla a veces es bastante compleja, lo que supone un decremento de prestaciones en cuanto al rendimiento de la aplicación, ya que en todo momento se tendrá que evaluar si se produjo una colisión entre el rayo, esfera o elipse, según la técnica empleada, y la malla en cuestión.

En tales casos puede ser útil la utilización de una malla especifica de colisiones, que simplifique la malla que caracteriza la forma del objeto 3D.

La implementación de esta técnica pasa por la herencia entre objetos que se ha explicado en apartados anteriores. Para llevarla a cabo existirá un objeto padre caracterizado por dicha malla de colisión del que heredan los demás objetos en los que se encontrarán deshabilitadas las colisiones. De este modo, el objeto padre se encontrará externo al mundo por lo que no será renderizado, pero en todo momento se evaluarán las posibles colisiones en las que se verá inmerso.

2.4. - Otras tecnologías involucradas en el proyecto

2.4.1. - Java

Java está relacionado con C++ que a su vez es un descendiente directo de C. La mayor parte de la estructura de Java está heredado de estos dos lenguajes, de C ha tomado su sintaxis mientras que la mayoría de las características de la programación orientada a objetos han sido tomadas de C++.

La creación de Java está íntimamente relacionada con el proceso de refinamiento y adaptación que se ha estado produciendo en los lenguajes de programación y que se viene dando desde los años 70. Cada innovación característica del diseño de este lenguaje viene fomentada

por la necesidad de resolver algún tipo de carencia básica que los lenguajes precedentes eran incapaces de resolver [1].

2.4.1.1. - Creación de Java

La época de finales de los 80 y principios de los 90 se caracterizó por el control que impuso la programación orientada a objetos y C++, llegando incluso a parecer, durante algún tiempo que se había llegado a un lenguaje perfecto. C++ había sabido mezclar la alta eficiencia y elementos característicos del estilo C junto con el paradigma de orientación a objetos, llegando a un leguaje útil en la creación de una gran variedad de programas.

Sin embargo surgieron fuerzas que impulsaban a la creación de un nuevo lenguaje de programación, en concreto la gran expansión de la WWW. Este nuevo lenguaje sería Java.

Al contrario de lo que pensaba mucha gente, el impulso inicial de Java no estuvo basado en Internet. La motivación principal de la creación de este nuevo lenguaje se debió a la necesidad, cada vez más latente, de contar con un lenguaje capaz de crear programas que se ejecutasen de manera independiente de la plataforma que se encontrase debajo, y por lo tanto de forma independiente de la plataforma que se hubiese utilizado para crear dicho software.

Esta es la semilla de Java, pero tuvo que pasar algo más para hacer de este lenguaje una herramienta tan extendida y útil, este algo fue la explosión de la WWW. Si el mundo Web no hubiese sufrido tal expansión a la vez que se desarrollaba Java, este lenguaje se hubiese quedado destinado a convertirse en un lenguaje oscuro orientado a la programación de dispositivos electrónicos para el consumo (que eran los que hasta ahora requerían soportar multiplataforma). Sin embargo esta explosión, puso en evidencia la necesidad de contar con programas portables, lo que demostraba la necesidad de un lenguaje de programación que soportase ser ejecutado desde diversas plataformas.

De esta forma se desarrollo Java, como una fusión entre la sintaxis de C y la orientación a objetos propia de C++, mejorando y refinando el paradigma de orientación a objetos utilizado por este último [1][2].

2.4.1.2. - Código Binario

Como ya se ha dicho, Java se crea como un lenguaje de programación que aspira a resolver los problemas con que cuentan los lenguajes que existían hasta la fecha, relacionados con la seguridad y portabilidad.

Finalmente se consigue solventar tales problemas haciendo que la salida obtenida tras compilar un código fuente determinado deja de ser un código ejecutable, como hacían sus predecesores, para pasar a ser un código binario (bytecode). Este código binario se encuentra formado por un conjunto de instrucciones altamente optimizado, diseñado para ser ejecutado por una máquina virtual que emula el intérprete de Java, siendo éste un intérprete de código binario.

Debido a que los programas Java son interpretados en vez de compilados, resulta más fácil llevar a cabo una ejecución en una gran variedad de entornos, debido a que solo es necesario implementar el intérprete adecuado para cada plataforma. De este modo, una vez exista el programa de ejecución para un sistema dado, será posible ejecutar cualquier tipo de programa Java desde dicha plataforma.

Otra de las prestaciones consecuencia de esto, a la par de la portabilidad, reside en la seguridad. Como la ejecución de los programas Java se encuentra controlada por el intérprete correspondiente, este último realizará labores de contención, evitando que no se produzcan efectos no deseados en el sistema en caso que sea necesario [1] [2].

2.4.1.3. - Factores característicos de Java

Es importante mencionar que, aunque la seguridad y portabilidad fueron los factores principales que desencadenaron el desarrollo de este nuevo lenguaje, se trabajó en implementar otros factores de igual relevancia que forman un papel crucial en el producto final obtenido. Así por lo tanto, los factores característicos de Java además de la portabilidad y seguridad, son:

- *Simple*: Fácil de aprender y utilizar.
- *Orientado a Objetos*: Se basa en una aproximación limpia, útil y pragmática de los objetos, de modo que ha llegado a un punto intermedio entre el modelo purista (todas las cosas son objetos), y el pragmático. Como resultado tenemos un modelo de objeto simple y fácil de ampliar.
- *Robusto*: Java proporciona fiabilidad, obligándose a encontrar pronto los errores que se producen en el desarrollo del programa.
- *Multi-hilo*: Permite escribir programas que puedan contar con varias líneas de ejecución, proveyendo de herramientas para gestionar el orden en el que se ejecutan cada una de estas líneas.
- *Arquitectura neutral*: Característica que dota de longevidad y portabilidad al código. En el momento de su creación se optó por tomar decisiones tanto en el diseño de su estructura como en la de su intérprete para que el producto desarrollado fuese capaz de mantenerse vigente a pesar de las actualizaciones de sistemas operativos, procesadores, cambios en los recursos básicos del sistema, etc.…
- *Interpretado*: Java se basa en la interpretación del código fuente compilado (bytecode), pudiéndose hacer ésta desde cualquier plataforma que tenga un intérprete Java.
- *Alto rendimiento*: Java se encuentra diseñado para ser ejecutado de manera eficiente desde cualquier CPU. Para ello se tuvo cuidado en diseñar un código binario fácil de traducir directamente al código máquina nativo propio de una máquina, consiguiendo así un alto rendimiento.
- *Distribuido*: Java fue enfocado para trabajar en un entorno distribuido como es Internet, por lo que trabaja con el protocolo TCP/IP. Esto se puede apreciar en que un acceso a un recurso por medio de un URL no difiere del acceso que se puede hacer a un archivo.
- *Dinámico*: Los programas propios de Java llevan implícita cierta cantidad de información utilizada para la verificación y resolución de los accesos a objetos en tiempo de ejecución, característica que permite enlazar dinámicamente el código de forma segura y adecuada.

2.4.2. - XML (eXtensible Markup Language)

XML es un meta-lenguaje extensible de etiquetas desarrollado por el World Wide Web Consortium, que permite definir lenguajes de marcado adecuados a diferentes usos.

Hoy en día cuenta con un papel importantísimo en el ámbito de la compatibilidad entre sistemas, ofreciendo las herramientas necesarias para una puesta en común de información caracterizada por su fiabilidad, facilidad y seguridad.

XML se define como un lenguaje para la definición de gramática de lenguajes específicos, de modo que los elementos que lo componen pueden proporcionar información acerca de lo que contienen y no necesariamente tan sólo de su estructura física o presentación.

Por ello, éste surge como un lenguaje de bajo nivel (nivel de aplicación) que posibilita el intercambio multiplataforma de información estructurada, pudiéndose utilizar desde cualquier aplicación que queramos, como bases de datos, editores de texto, hojas de cálculo, etc.

XML surge como una tecnología sencilla complementada por otras que tiene a su alrededor, que la hacen mas grande y la dotan de mayores posibilidades [10]].

2.4.2.1. - Historia

XML tiene sus raíces en el lenguaje GML(General Markup Languaje), inventado por IBM en los años 60 debido a la necesidad de almacenar grandes cantidades de información. Este lenguaje evolucionó para convertirse en el llamado SGML (Estándar General Markup Languaje) a partir del que se crearon diversos sistemas para el almacenaje de información.

Seguidamente, en el año 89 se desarrollo el lenguaje HTML, ampliamente difundido por la WWW. HTML ha ido creciendo de forma descontrolada, ignorando las reglas y etiquetas impuestas por el W3C destinadas a la estandarización del lenguaje. Finalmente en 1998, el W3C inició el desarrollo de XML en el que aún sigue inmerso, con la finalidad de crear un lenguaje común a numerosos lenguajes de programación.

2.4.2.2. - Objetivos del desarrollo de XML

Los objetivos que se marcaron en el desarrollo de XML fueron varios, entre los que se destacan:

- Contar con una herramienta caracterizada por tener un carácter idéntico a la hora de servir, recibir y procesar la información HTML, de modo que se pudiese aprovechar toda la tecnología vigente implantada.
- Tener un carácter normal y conciso, desde el punto de vista de los datos y su manera de guardarlos.
- Necesidad de contar con un carácter extensible, de modo que se pudiese utilizar en todos los campos del conocimiento.
- Facilidad a la hora de leer, editar e implementar.
- Facilidad a la hora de implantar, programar y aplicar a sistemas caracterizados por diversas propiedades.
- Compatibilidad total en la comunicación de datos. Cualquier información transmitida en XML podría ser recibida y comprendida por cualquier otra aplicación diferente.
- Posibilidad de realizar una migración de datos. Si se trabaja en formato XML será muy sencillo hacer un traspaso de datos de una base de datos a otra.
- Versatilidad a la hora de desarrollar aplicaciones Web. Utilizando XML únicamente existiría una sola aplicación encargada de manejar los datos, contando con una hoja de estilo o similar para cada navegador, encargada de aplicarle el estilo adecuado.

2.4.3. - JDOM (Java Document Object Model)

Se trata de un documento de modelado de objetos de código fuente abierto diseñado específicamente para su utilización bajo Java, tomando así las ventajas propias de este lenguaje.

Esta librería constituye una API simple pensada para la lectura, manipulación y creación de documentos XML, característica común que comparte con las especificaciones DOM y SAX. Aunque existen muchas diferencias entre ambos, ya que mientras DOM y SAX fue concebido

como un lenguaje neutral e inicialmente usado en la manipulación de documentos HTML con JavaScript, JDOM se ha creado específicamente para su utilización en Java, constituyendo así una extensión mas natural, intuitiva y correcta que las aproximaciones existentes antes de su concepción [10].

2.4.3.1. - Estructura de JDOM

A continuación se describirá a grandes rasgos la estructura de la API JDOM. De ésta se puede decir que se encuentra compuesto por los siguientes 5 paquetes:

- *Paquete 'org.jdom'*: Destacan las clases 'Document' encargada de la representación de un documentos XML, 'Element' que representa a cada uno de los elementos o etiquetas que constituyen el documento, y 'Atribute' que hace referencia a los atributos propios de un elemento.
- *Paquete 'org.jdom.adapters'*: Engloba a todas las clases que implementan los adaptadores utilizados para adecuar su comportamiento a la API propia del analizador sobre el que se encuentra instalado, ya que no todos los analizadores DOM cuentan con la misma API.
- *Paquete 'org.jdom.input'*: Engloba todas aquellas clases involucradas en la creación de un documentos XML
- *Paquete 'org.jdom.output*': Alberga todas las clases involucradas en la salida de nuestra clase Document.
- *Paquete 'org.jdom.transforms*': Engloba a todas las clases encargadas de realizar transformaciones con tan solo una línea de código. Todas éstas están basadas en las clases JAXP TrAX.

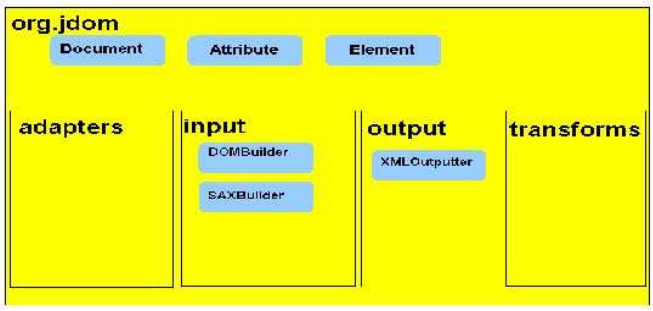

Figura 18: Estructura de JDOM

3.- Análisis

En este apartado se lleva cabo el análisis del funcionamiento de la plataforma diseñada. Para ello se desglosa su estructura en cada una de las partes que la componen, desde el nivel más bajo hasta la interfaz que implementa la iteración con el usuario.

Primeramente, y para que sirva de toma de contacto, es preciso puntualizar que esta plataforma busca ser una solución útil, intuitiva y fácil de utilizar a la hora de crear juegos (en un principio destinados a la educación aunque esta definición se puede extender a cualquier clase de juegos).

3.1.- Punto de partida

Desde un principio se trató de implementar una herramienta con gran potencial, capaz de ser la base de juegos de una versatilidad enorme en cuanto a su complejidad. Para ello, y con la finalidad de impulsar la realización de aplicaciones complejas, la labor de creación se separa en dos. Por una parte está el papel del desarrollador, rol encargado del diseño del guión del juego. Su labor implicaba un conocimiento suficiente de XML así como de la estructura de la máquina de estados de la que se sirve el 'core' de la aplicación a la hora de implementar el juego.

Por otro lado se encuentra el rol del docente, personaje encargado de desarrollar el contenido didáctico de la aplicación, que pasaba al desarrollador en modo de archivos XML, para su correcta inclusión en el juego. De este modo, el tutor únicamente será el encargado del desarrollo de los documentos que contengan tanto el temario que se quiere impartir así como los tests que se quieren plantear, de forma que el desarrollador pueda hacerles referencia desde el guión del juego creado.

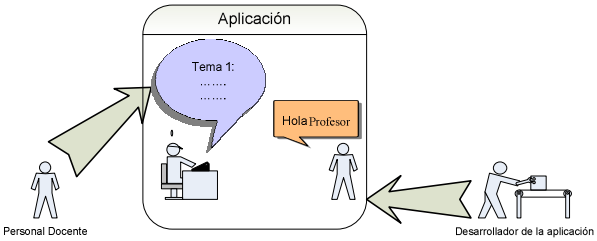

Figura 19: Dualidad en el diseño

Para permitir la facilidad de uso por parte de los desarrolladores se optó por la utilización de documentos XML. Para el manejo de estos documentos se empleó la API JDOM que permite trabajar desde Java en la creación y análisis de dichos archivos. Esta herramienta facilita de una forma asombrosa la descomposición del guión del juego y de sus archivos auxiliares.

En cuanto a la infraestructura utilizada por la plataforma, se optó por un motor gráfico que aporta suficiente potencia como para crear juegos de muy diversa índole, evitando caer en una dinámica de juegos cuya característica más destacable sea su similitud. Por tal motivo, además de su implementación en Java (lo que conlleva la aportación de otras características buscadas como orientación a objetos, soporte multiplataforma, facilidad de uso, librerías muy variadas que contienen herramientas muy útiles en la implementación del núcleo de la plataforma como JDOM), se consideró el motor gráfico jPCT como una herramienta óptima para formar parte de la infraestructura del sistema.

La plataforma puede tener objetos inertes, ascensores y/o personajes.

Estos personajes pueden tener diálogo (compuesto por varias intervenciones). Cuando el usuario colisionaba con un personaje con diálogo la aplicación mostraba el contenido de la intervención actual.

Las intervenciones podían ser de varios tipos: texto, con opciones, temas o tests.

Si la intervención era de tipo tema, la aplicación mostraba el contenido de un fichero XML identificado por un id.

Por el contrario, si era de tipo test, la aplicación mostraba una serie de preguntas tipo tests contenidas en otro fichero XML.

A continuación se indica el estado inicial de la aplicación:

- La primera versión era una aplicación en la que se ejecutaba después de que estuvieran hechos los ficheros XML docentes (temario y tests) y el guión de juego.
- La única ayuda al desarrollador del juego que te daba era la posibilidad de saber las coordenadas en las que se encontraba la cámara/protagonista, para facilitar la ubicación de los componentes del juego. Para ello teníamos que haber pasado como parámetro la opción '-depuración' y mostraba las coordenadas siempre y cuando el usuario pulsara la tecla 'd'.
- Dicha aplicación no reproducía audio. Por lo que el juego no tenía la opción de sonido, ya fuera sonido ambiente o sonido de diálogo.
- La conversación de un personaje no tenía la posibilidad de grafo, es decir cuando se llegaba a una intervención de tipo opción dicha conversación tenía tantas bifurcaciones como opciones presentaba, por lo que eligiendo un camino no cabía la posibilidad de llegar al mismo punto que otra opción distinta dentro de la misma conversación (la única posibilidad que se daba sería escribir dos (o más) intervenciones iguales para simular que se había llegado al mismo camino por distintas opciones).
- En el caso de encontrarse con un personaje con diálogo, al colisionar con dicho personaje aparecía una ventana emergente que mostraba el texto de la intervención, estando ésta fuera de la aplicación.

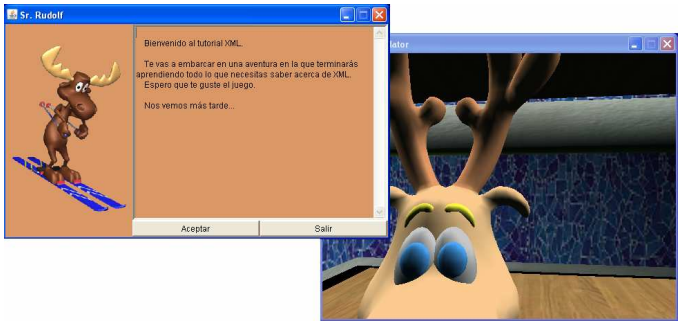

Figura 20: Ejemplo de ventana de diálogo en la versión 1

- Los personajes en la primera versión no se movían de su sitio, estaban estáticos, como mucho, el único movimiento que realizaban era una rotación sobre su eje Y, es decir, tenían un giro con un ángulo determinado sobre el eje ya mencionado.
- Los exámenes sólo daban la posibilidad de elegir las respuestas dentro de una serie de opciones dadas, es decir sólo podían ser de tipo test. Permitían la posibilidad de tener varias preguntas, pero todas con el mismo número de opciones, no se podía tener una pregunta con X opciones y otra pregunta dentro del mismo test con X+1 opciones.

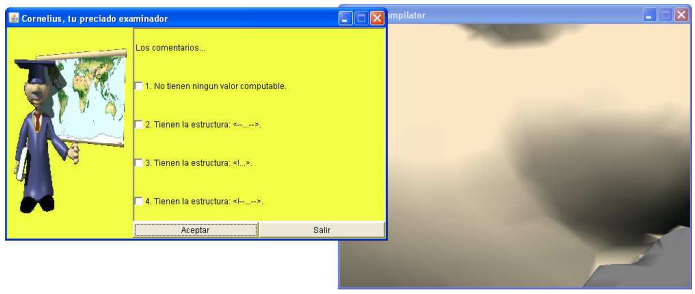

Figura 21: Ventana tipo test
(todas las preguntas de este test tenían 4 opciones de respuesta)

3.2.- Situación deseada

A continuación se describirá la evolución planteada en la plataforma:

3.2.1. Conversación

Cualquier juego diseñado para esta plataforma asienta sus bases en el diálogo con los personajes. Estas conversaciones son las que proporcionan los medios suficientes para pasar a una nueva fase, permitiendo progresar en el juego.

Para poder iniciar un diálogo es necesario colisionar con el personaje en cuestión. El motor detectará dicho evento parando así la ejecución del programa y lanzando una ventana de diálogo en la que se podrá conversar con él.

Las intervenciones que integran toda conversación pueden ser de distintos tipos según su contenido:

- *Modo texto*: Intervención basada en un diálogo (puede ser también exposición del tema docente).
- *Modo opción*: Intervención en la que se ofrece al protagonista una bifurcación de 2 a 5 ramas que marcarán el devenir de la conversación, así como de su avance en el juego.
- *Modo test*: Intervención en la que plantea al protagonista una prueba en forma de test multi-respuesta. El resultado de esta prueba marcará el devenir de la conversación así como del juego.

Una vez finalizada la conversación, y dependiendo del carácter de la misma, el usuario continuará en la misma fase en la que se encontraba o bien pasará a otra diferente a la que le condujo la anterior iteración. En cualquiera de los dos casos se inhabilitará la posibilidad de hablar con el último personaje durante 5 segundos, evitando así caer en un bucle de colisiones infinitas.

En la versión anterior, no había la posibilidad de que dos o más opciones del mismo diálogo volvieran a encontrarse a lo largo de la conversación. Esta versión se ha mejorado para que esto fuera posible y así no fuera necesario crear distintas intervenciones con idéntico tipo y contenido para simular que comparten conversación (cuando llegara la simulación de parte común, en vez de tener ese camino común tendríamos X caminos idénticos).

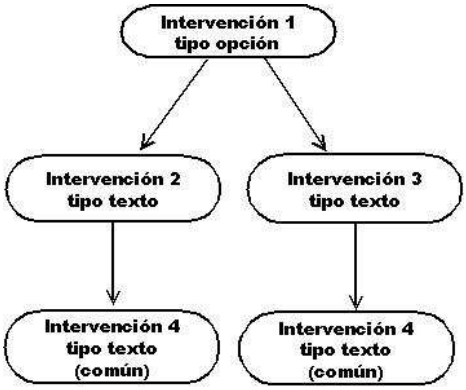

Figura 22: Representación sin implementación de parte común simulando una parte en común

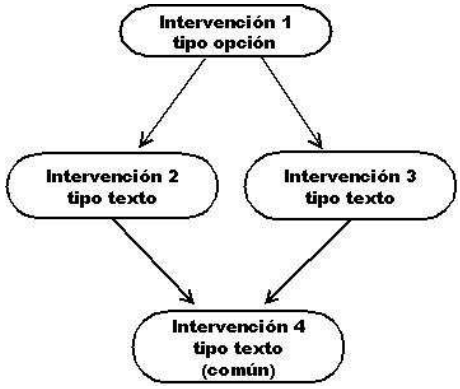

Figura 23: Representación con implementación de parte común

Antes se ha dado una pequeña pincelada de las intervenciones existentes en una conversación (las que contienen texto), pero existen más dependiendo de su carácter:

- ***Intervención Texto***: Caracterizada por un contenido de texto y representado en un cuadro de texto en la ventana lanzada. Este nodo contará con una sola intervención hija perteneciente a los tipos aquí mencionados.

- ***Intervención Tema***: Parecida aunque no igual a la intervención Texto. Se basa en los mismos principios aunque se distingue de la anterior en que, mientras la primera recibe su contenido del archivo XML que define el guión del juego, ésta recoge su contenido de un archivo XML diferente, creado por el profesor con material docente.

- ***Intervención Test***: Intervención destinada a la presentación de una prueba con la que evaluar los conocimientos del usuario durante el transcurso de una conversación. Un objeto de este tipo contiene entre sus atributos una batería de preguntas elegidas al azar del archivo XML especificado, así como la nota de corte para poder suspender o aprobar el examen.

  En lo que refiere al número de opciones ofrecidas el sistema permite entre 2 y 5. En caso de que el test soporte una única posibilidad, la ventana que se mostrará no será la de elección de respuesta sino la que permite al usuario introducir dicha respuesta por teclado (algo nuevo para la aplicación).

  Por lo tanto, un diálogo vendrá caracterizado por un número concreto de respuestas. Este parámetro será comprobado en todo momento por el depurador de la aplicación de modo que, si se detectase la aparición de diferentes archivos XML de test con un número diferente de respuestas, se procederá a anulas la síntesis del juego y comentar dicho error en la consola.

  En lo que a descendientes se refiere, este tipo de intervención contendrá otras dos intervenciones hijas relacionadas con la superación o no de la prueba (aprobado o suspenso).

- ***Intervención Opción***: Nodo del árbol conversacional que hace las veces de bifurcación en tantas ramas como se concreten en el guión del juego. Este nodo está caracterizado por una etiqueta que implementa la proposición planteada y por un número, mayor o igual que uno, de etiquetas que describan las distintas posibilidades que se le ofrece al usuario.

  En cuanto a los descendientes, se pueden distinguir tantos como opciones se definan dentro de esta intervención.

- ***Intervención Final***: Entidades de este tipo implementan las hojas del árbol que define una conversación. Son los nodos finales encargados de indicar la fase siguiente a la que se debe dirigir el juego en caso de haber llegado a dicho punto dentro de la conversación.

  Como tal, en el guión del juego se indicará el identificador de dicha fase, comprobándose posteriormente en el depurador la existencia de la misma. Si el valor de

éste es ‘-1’ supondrá que no existe cambio de estado alguno, en caso contrario se procederá a la configuración de dicho estado siguiente en el controlador del juego.

- ***Intervención Común***: Entidades de este tipo son los nodos encargados de indicar la parte común de la conversación con la que se enlaza dicha intervención en caso de haber llegado a dicho punto dentro de la conversación (como se ha comentado anteriormente este tipo de intervención es una evolución de la plataforma).

  Por lo tanto, cuando aparezca esta intervención se enlazará directamente con otra que será compartida por otra/s intervención/es. Como conclusión decir que este tipo de intervenciones contará con una sola intervención hija.

- ***Intervención Parte_Comun***: Son las intervenciones que se comparten dentro de una conversación. Este nodo contará con una sola intervención hija perteneciente a los tipos aquí mencionados (al igual que la intervención anterior, esta intervención también es nueva para la plataforma).

3.2.2. Tipos de tests

En el apartado 3.1. se ha comentado el tipo de test que soportaba la aplicación dejando ver que con esta segunda versión se pretende mejorar el soporte de varios tipos de exámenes.

Así se puede decir que ya un test no tiene por qué tener preguntas con el mismo número de opciones cada una de ellas. En esta ocasión se pueden ver por ejemplo, preguntas con tres opciones, otras con cuatro opciones, etc. dentro de un mismo test.

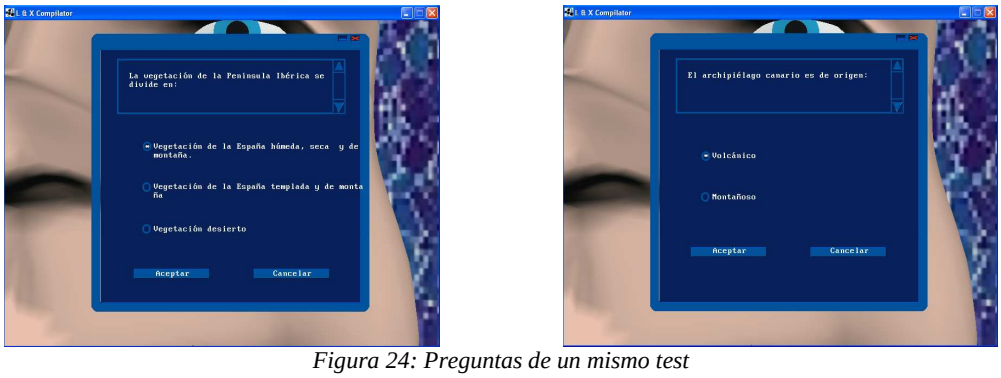

Figura 24: Preguntas de un mismo test

Además, la versión anterior sólo soportaba respuestas tipo test (donde daba una serie de opciones de las que se tenía/n que elegir la/s respuesta/s correcta/s). En esta mejora se pueden ver preguntas que pidan introducir la respuesta por teclado (respuestas libres). Este tipo de preguntas se recomendarían para aquellas preguntas que sólo tienen una respuesta válida (y sin poder darse con distintas palabras), como por ejemplo operaciones matemáticas, etc.

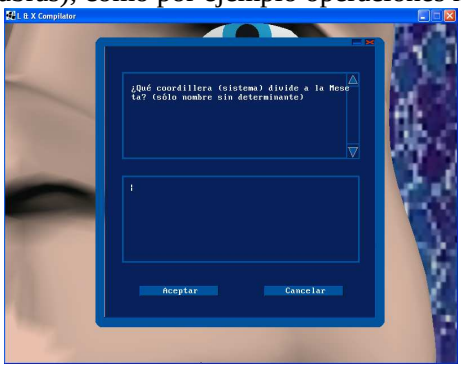

Figura 25: Pregunta de libre respuesta

Por otro lado, al igual que la versión inicial, las preguntas tipo test pueden ser de una sola respuesta o multirrespuesta válida.

Todos estos tipos de preguntas son soportados por un test/examen, pudiendo estar mezcladas, por lo que no es obligatorio que un examen sólo tenga preguntas tipo test o sólo preguntas de respuestas libres.

3.2.3. Ventanas

Como se dijo en el apartado 3.1, la versión inicial tenía ventanas emergentes donde mostraba las intervenciones de los diálogos de los personajes.

En este caso, dichas ventanas han sido modificadas para que fueran incrustadas dentro de la aplicación.

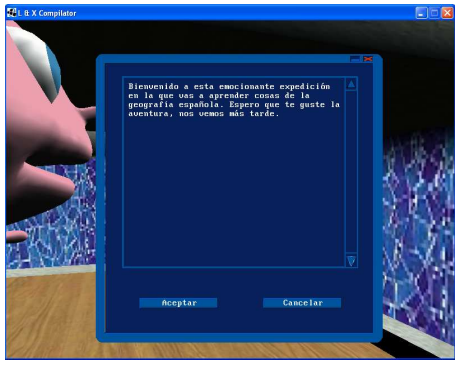

Figura 26: Ventana de diálogo dentro de la aplicación

Así un objeto ventana es una imagen (en este caso de extensión .gif) que se muestra dentro del motor gráfico jPCT.

Figura 27: Tipo de ventana

Para poder representar el texto dentro de la aplicación, cada letra que se muestra dentro del motor gráfico es una porción de imagen que contiene todas las letras que pueden ser impresas por la aplicación.

Figura 28: Caracteres que imprime la aplicación

De esta forma, una ventana puede contener etiquetas, campos de texto y botones.

Los campos de texto son los objetos necesarios para poder introducir datos en la aplicación.

Los botones pueden mostrar o no etiquetas identificativas. Cada botón puede tener un manejador de eventos que determine las acciones que se llevarán a cabo tras presionar dicho botón.

3.2.4. Movimiento de personajes

Una diferencia clara con la aplicación inicial es que si en la versión anterior los personajes carecían de movimiento, o en su defecto el único movimiento que realizaban era un pequeño giro sobre su eje Y, en esta versión tendrán la posibilidad de mantener el movimiento

anterior descrito además de poder realizar un recorrido dinámico. Así, esta aplicación tendrá una animación y no se quedará en una aplicación con personajes estáticos.

Así un personaje que tenga determinado un movimiento (recorrido) deberá indicar sobre qué eje caminará paralelamente (X o Z), así como su posición final (distancia desde el origen). Por lo que tendrá una implementación XML del estilo:

```
<Personaje posicion_x="390" posicion_y="-49" posicion_z="-600" directorio_texturas="juegos-Geografia-3ds-personaje-fig2" id="0"
archivo_3ds="juegos-Geografia-3ds-personaje-fig2-antfree.3ds" escala="2.3f" rotacion_y="2.3f" velocidad="0.025" eje="x" posFinal="100" />
```

Mientras que un personaje que sólo realice un giro sobre su eje Y tendrá una implementación XML del estilo:

```
<Personaje posicion_x="390" posicion_y="-49" posicion_z="-600" directorio_texturas="juegos-Geografia-3ds-personaje-fig2" id="0"
archivo_3ds="juegos-Geografia-3ds-personaje-fig2-antfree.3ds" escala="2.3f" rotacion_y="2.3f" movimiento="si" velocidad="0.025"/>
```

Por otro lado, la implementación de un personaje que no realice ningún tipo de movimiento podrá ser:

```
<Personaje posicion_x="390" posicion_y="-49" posicion_z="-600" directorio_texturas="juegos-Geografia-3ds-personaje-fig2" id="0"
archivo_3ds="juegos-Geografia-3ds-personaje-fig2-antfree.3ds" escala="2.3f" rotacion_y="2.3f" movimiento="no" />
```

Para que un personaje tenga animación en su caminata, antes se deberán haber generado los fotogramas necesarios para simular dicho movimiento. Así de la imagen de ese personaje se debe sacar por lo menos 4 imágenes más, una en la que adelante un poco una pierna, otra en la que adelante más la pierna de antes y la otra la retrase un poco, y las otras dos imágenes similares a éstas pero con la otra pierna [12].

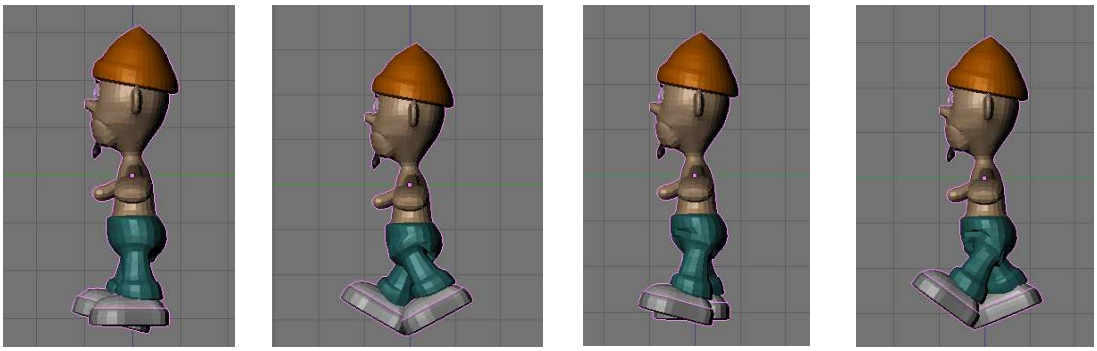

Figura 29: Imágenes 3D para la animación de andar

Cuantos más fotogramas se saquen, más larga y de mejor calidad será la animación. Así a la vez que se mueven las piernas también se podrían mover los brazos por ejemplo.

3.2.5. Ascensor

Un ascensor es un objeto en el mundo cuya función es facilitar al protagonista poder subir (o bajar) a otra planta del escenario.

Para ello, un ascensor tiene distintos modos de funcionamiento, los cuales pueden ser:

- ***Modo Alternativo***: El ascensor se moverá únicamente si el protagonista está colisionándolo o si se encuentra ubicado en una posición que no sea ni la primera planta ni la segunda. En otro caso permanecerá inmóvil.
- ***Modo Movimiento***. El ascensor se encuentra en continuo movimiento, subiendo y bajando alternativamente.
- ***Modo Planta Uno***: El ascensor permanece inmóvil en la planta uno mientras que no se produzca una colisión, caso en el que se desplazará a la planta dos para posteriormente volver a recuperar su posición inicial.

Modo Planta Dos: Similar al modo anterior, pero en este caso la posición inicial será segunda planta.

3.2.6. Sonido

Como ya se ha dicho, la primera versión carecía de sonido. Esta segunda versión se ha implementado de tal forma que pueda soportar sonido. Así está la posibilidad de que se reproduzca o no cuando el usuario esté jugando.

Se pueden destacar dos tipos de reproducciones:

- *Diálogo*: reproducido siempre que esté activo el sonido y el usuario se encuentre con un personaje con diálogo. Siempre que el personaje no tenga una intervención de tipo test (las preguntas pueden ser aleatorias), cabe la posibilidad de que el sonido que se reproduzca corresponda con el texto del diálogo, por lo que este tipo de sonido, sólo se reproducirá una vez por intervención.
  Los archivos reproducidos por este tipo de sonido serán definidos en cada intervención.
  En caso de que la intervención no tenga definido un archivo de sonido, siempre que se encuentre con una intervención se reproducirá un sonido por defecto.
- *Sonido ambiente*: reproducido siempre que esté activo el sonido y no se esté en una intervención (diálogo). Este sonido será reproducido de forma infinita (cuando se esté jugando) siempre que no haya una conversación activa.
  El archivo reproducido por esta intervención debe ser llamado “ambiente.wav” y estará contenido, como es debido, dentro del directorio sonidos.

3.2.7. Diseño

En este punto se pueden destacar tres “aplicaciones”: Docencia, Edición y Juego.

El apartado Docencia corresponde al desarrollador del contenido docente para generar los archivos XML docentes, Edición corresponde al desarrollador del juego para generar el guión del juego, mientras que Juego será la aplicación final para que el niño/usuario pueda jugar a la vez que aprender (siempre que dicho juego sea didáctico).

A continuación se explicará detalladamente las características de cada una de las nuevas pequeñas aplicaciones que tiene esta versión.

- *Juego*

Si se observa la aplicación desde el punto de vista del usuario final, la tarea principal que caracteriza el uso de la plataforma por parte del mismo está basada en su participación en el entorno virtual diseñado, participando así del juego desarrollado.

Las características de la aplicación hacen que las posibilidades del jugador en su iteración con el sistema puedan ser de tres tipos: moverse a través del escenario definido, conversar con los personajes ubicados en la escena y salir de la aplicación. Estos tres principios implementan las posibilidades que se le ofrece al usuario de todo juego diseñado con esta plataforma.

Así pues, las aplicaciones finales a las que se va a llegar están basadas en juegos que ubican al usuario en un entorno virtual en el que convivirá con diferentes personajes, ascensores y objetos inertes, y en el que puede progresar según va dialogando con los personajes adecuados.

Dichos personajes, en sus diálogos con el jugador, pueden exponer conversaciones sencillas o complejas, bifurcaciones de varias ramas que caracterizaran el devenir del protagonista (que luego pueden converger en una rama) y pruebas en forma de exámenes que le enviarán a una fase determinada según el nivel que refleje.

Una de las mejoras propuestas para la evolución de la plataforma es la posibilidad de guardar una partida. Para ello, el usuario debería pulsar la tecla ‘g’ y dar el nombre a la partida (donde en caso de no empezar por ‘juego’ se añadirá este prefijo al nombre dado).

Por lo que una vez dado el nombre se generará un nuevo guión de juego idéntico al utilizado en la partida, con la única diferencia de la fase inicial y la posición del protagonista (que serán los datos que había en la partida).

Así entonces se puede elegir el guión de juego a utilizar por la plataforma desde la aplicación (en la primera versión sólo se podía elegir modificando los parámetros del archivo ejecutable .bat).

Por lo tanto la iteración del usuario con la aplicación final puede ser: elegir archivo de juego, mover al protagonista, hablar con los personajes, guardar el juego y salir de él.

- *Docencia*

Como se ha comentado anteriormente, el encargado de desarrollar el contenido didáctico le pasaba al desarrollador del juego archivos XML donde se encontraban dichos datos.

Gracias a esta mejora, el docente no necesita saber absolutamente nada de XML puesto que no tiene por qué crear “a mano” dichos archivos. Desde esta aplicación puede ir introduciendo los datos docentes y posteriormente se generarán los archivos correspondientes.

Para ello, se parte de una ventana inicial donde se representan los datos “temas” y “test” que puede haber en el directorio raíz correspondiente y/o que acabamos de generar.

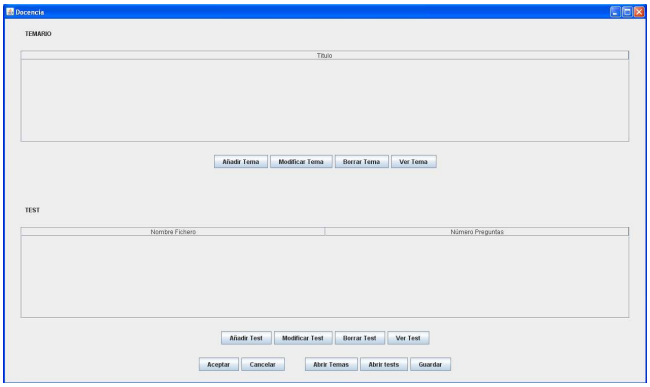

Figura 30: Ventana inicial docente

En dicha ventana se visualizan los datos en la tabla correspondiente. Así en caso de ser un tema se vería la cabecera correspondiente el identificativo del tema, mientras que en el caso de ser un test, en dicha cabecera lo que se vería sería el nombre del archivo junto con el número de preguntas que tiene dicho test.

En esta aplicación docente se puede añadir, modificar, borrar y visualizar los datos de un test o tema seleccionado anteriormente.

Para agregar una entrada nueva en la tabla, se debe pulsar el botón “Añadir Tema” o “Añadir Test” en función de lo que se quiera añadir. Si lo que se añade es un tema, se abrirá una ventana para introducir el contenido del tema.

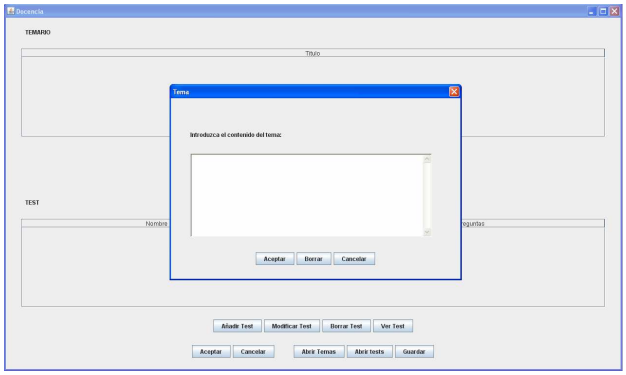

Figura 31: Ventana Tema

Si por el contrario, se quiere modificar un tema (o visualizarlo), se debe seleccionar la entrada correspondiente en la tabla y así mostrará la ventana anterior con el campo relleno según el contenido del tema seleccionado. En caso de que se esté visualizando, el campo no será editable.

Para crear un test nuevo, se debe hacer pregunta por pregunta, así en el caso de seleccionar esta acción se abrirá una ventana como la siguiente:

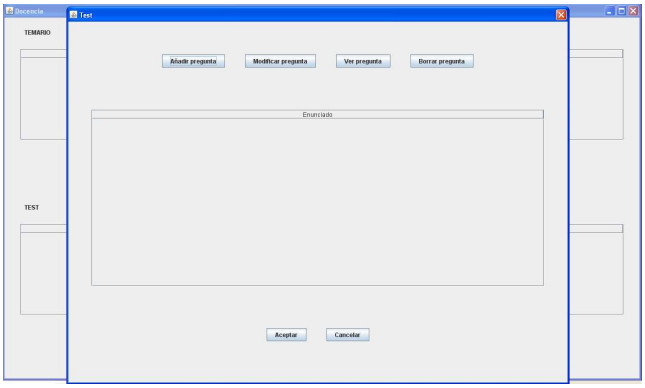

Figura 32: Ventana Test

En la ventana anterior se pueden visualizar los enunciados de las preguntas que tiene el test. Así cada vez que se añada o borre una, dicha tabla será actualizada.

Se tiene la posibilidad de añadir una nueva pregunta, modificar, borrar o ver una pregunta seleccionada anteriormente como en el caso de temas.

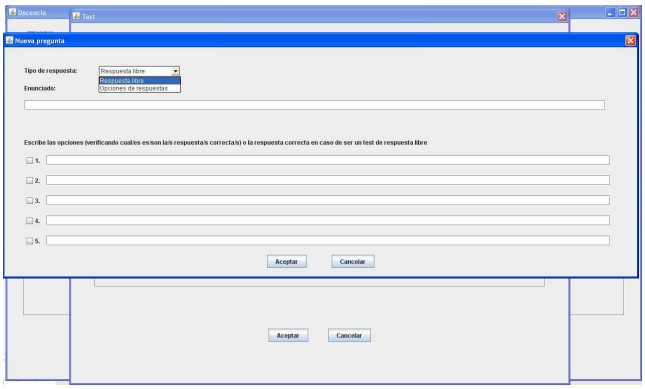

Figura 33: Ventana pregunta

Donde se puede de elegir el tipo de pregunta (no especificado en la primera versión):

- Respuesta libre: el usuario da la respuesta por teclado.
- Opciones de respuesta: el usuario debe elegir la/s respuesta/s correcta/s de una serie de opciones que tiene la pregunta.

Si una opción es una respuesta correcta, para indicarlo, y así que el test lo pueda saber, se debe activar la casilla de verificación correspondiente a dicha opción, en caso de no estar activa dicha casilla, la opción no será una solución de la pregunta.

Como ocurría en la parte de temas, si se visualiza una pregunta del test, los campos no serán editables, por lo que desde ahí no se podrían modificar los datos de la pregunta. Para modificarlos se debería pinchar, como es natural, en el botón “Modificar pregunta”.

Todos los datos correspondientes a los temas se guardan en un fichero llamado “curso+nombre que dé el desarrollador.xml” (si dicho nombre no empieza por curso), donde el creador puede seleccionar el directorio (debe ser una carpeta contenida en este directorio para que el fichero sea creado sin ningún problema con el DTD correspondiente).

Hay que decir que en el directorio donde se ha guardado el archivo correspondiente a los temas no puede haber varios ficheros que empiecen por “curso” ya que la aplicación detecta los archivos donde están almacenados los temas gracias a ese prefijo. Y en caso de

haber varios archivos que empezaran por ese prefijo, cuando se editara, la aplicación utilizaría forzosamente el primer archivo de este tipo que encontrara.

Cuando se cree o modifique un test (se seleccione “Aceptar” en la ventana test), la aplicación pedirá que se dé nombre al archivo para poder guardarlo posteriormente. Para nombrar un archivo test, si dicho nombre no empieza por “test” la aplicación añadirá esa palabra al inicio del nombre dado por el docente (para posteriormente poder identificar archivos que corresponden a un test).

Para finalizar este apartado, hay que destacar la posibilidad de poder modificar ficheros XML docentes creados anteriormente, para ello se debería pulsar en el botón “Abrir Temas” y/o “Abrir Tests” y así se visualizarían los datos correspondientes en las tablas.

Como se puede observar, este apartado de la aplicación no utiliza para nada el motor gráfico jPCT, son todas ventanas swing.

- *Edición*

Mientras que en la primera versión se tenía que hacer todo el guión del juego “a mano” editando toda la sintaxis XML necesaria, en esta nueva versión sólo es necesario hacer la implementación de unos determinados elementos (temporizadores, pantalla, escenario y protagonista) para la generación de un guión básico.

Así, el resto de elementos que puede tener un guión de juego pueden ser creados correctamente siguiendo el DTD correspondiente a través de la aplicación.

Por lo que, resumiendo, para poder editar un guión de juego con esta aplicación antes debe tener un guión básico donde se indiquen elementos básicos necesarios, tales como escenario, protagonista… y una vez que exista ese archivo XML básico se podrá utilizar la edición de juego que posee la aplicación.

Por tanto, para poder ir generando el resto de elementos XML que tendrá un guión de juego, la aplicación de edición irá pidiendo al desarrollador que vaya colocando los objetos y los diálogos (que pueden tener los personajes) por fases.

A continuación se muestran unos diagramas de flujo que representan la edición. Están separados por partes por claridad, así tenemos uno para colocar los objetos, otro para dar diálogo y el último para enlazar fases.

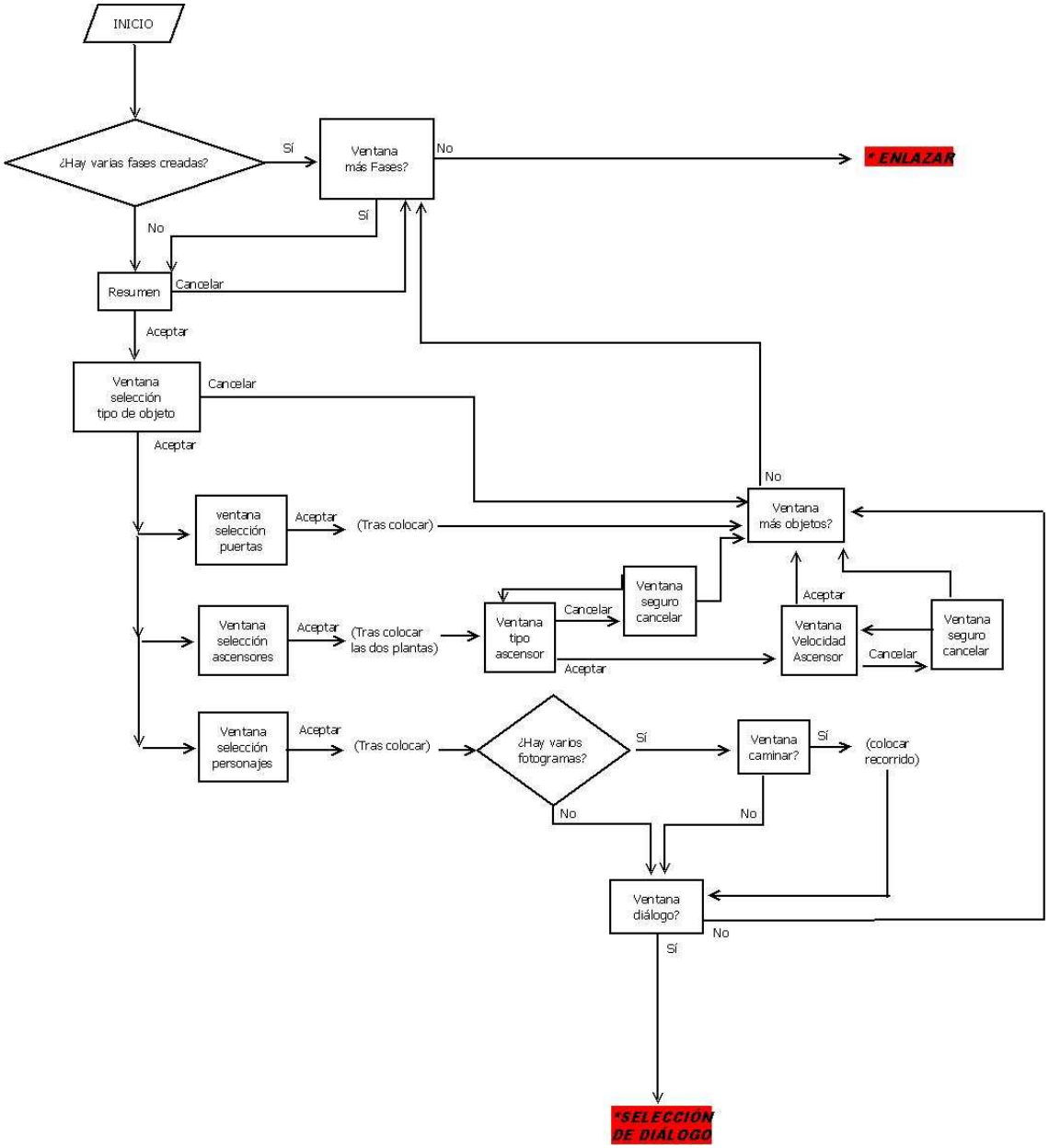

Figura 34: Diagrama de flujo para colocar objetos

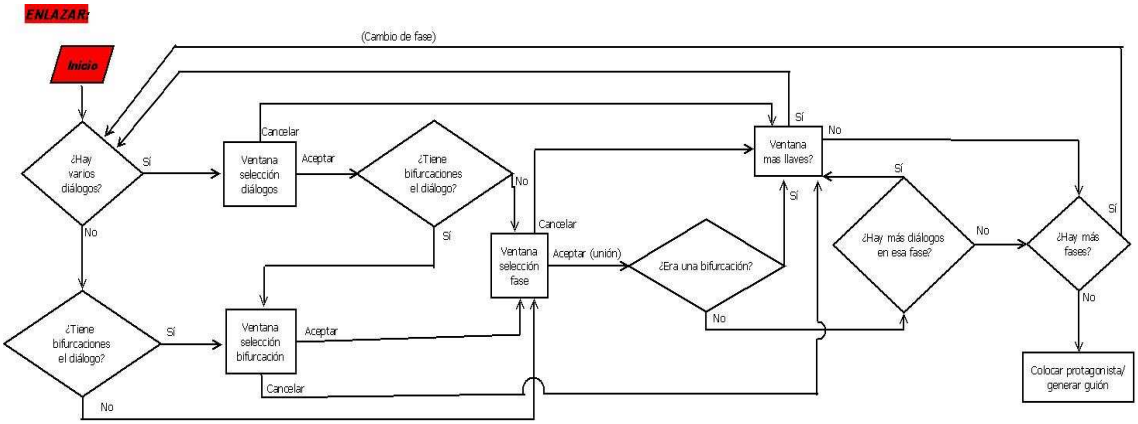

Figura 35: Diagrama de flujo para enlazar fases

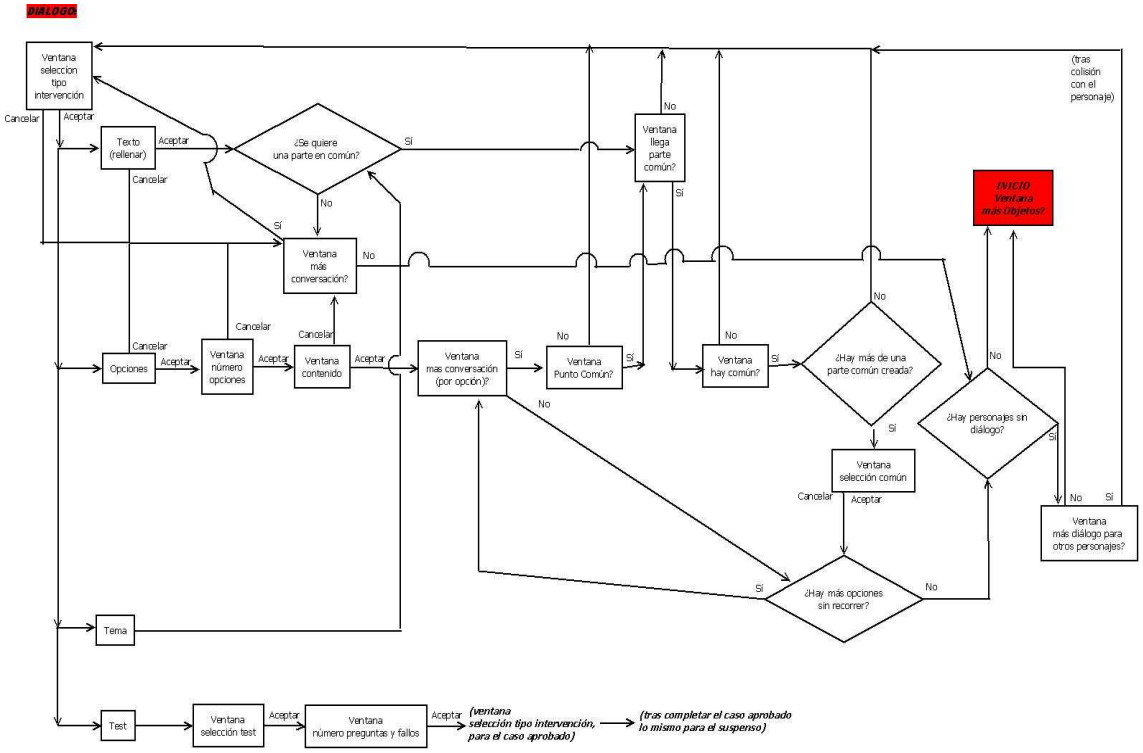

Figura 36: Diagrama flujo de diálogo

Para que la aplicación sepa de qué tipo de objeto posee, cada archivo .3DS debe ser colocado en su directorio correspondiente (puertas, ascensores, personajes, véase el directorio 'Geografia' que da de ejemplo la aplicación) (figura 34).

Para crear una fase a través de la aplicación, se debe tener en dicha fase al menos un personaje con un diálogo, si no hay diálogo la fase no será creada.

Para poder identificar una fase, lo primero que pide la aplicación es que el desarrollador introduzca un pequeño resumen de dicha fase. A continuación mostrará una ventana para seleccionar el tipo de objeto (puerta, ascensor, personaje) que se quiere colocar sobre el escenario.

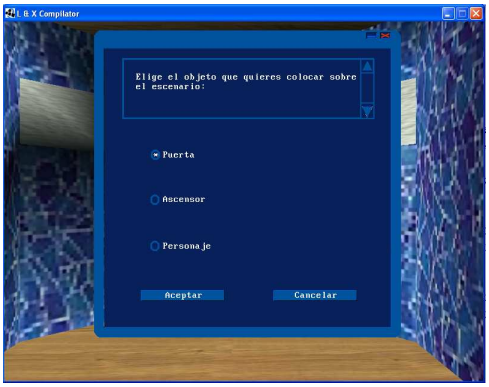

Figura 37: Ventana de selección de tipo de objetos

Una vez que se haya seleccionado el tipo de objeto que se quiere colocar sobre el escenario aparecerá una ventana para que el desarrollador pueda elegir el objeto 3D propiamente.

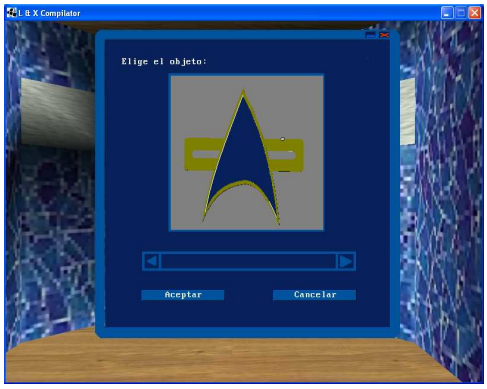

Figura 38: Selección de objetos

Como es lógico, el objeto mostrado en la ventana anterior sería el que desarrollador, en caso de aceptar, podría colocar en el mundo. Tiene la posibilidad, si hay más objetos de ese tipo (puerta, ascensor o personaje), de elegir otro distinto pulsando sobre el botón desplazamiento correspondiente para cambiar de objeto a elegir.

Una vez que se pulse sobre el botón aceptar, el desarrollador puede mover el objeto seleccionado por el escenario, cambiar la escala que tiene e incluso realizar giros sobre sus ejes antes de colocarlo donde desee y en la posición que quiera.

En caso de cancelar en la ventana anterior, la aplicación mostrará una ventana para asegurarse de que no se quiere colocar un objeto.

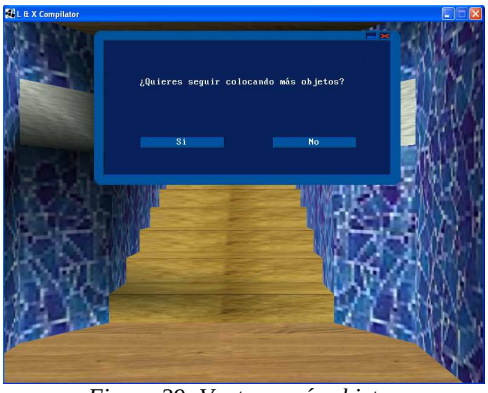

Figura 39: Ventana más objetos

La ventana anterior también será mostrada si se pulsa en cancelar en la ventana de selección de tipo de objetos.

En esta última ventana, si se pulsa sobre “Sí” volverá a aparecer la ventana de selección de tipo de objetos, mientras que si se pulsa en “No”, se guardarán los datos necesarios para crear los elementos XML correspondientes a la fase si dicha fase tiene al menos un personaje con diálogo, mientras que en caso contrario serán ignorados. Preguntará si se quiere crear más fases (donde en caso afirmativo volverá a pedir un resumen identificativo para la nueva fase y volverá a empezar, y en caso negativo si la aplicación tiene varias fases pedirá enlazarlas, que se verá más adelante; si no tiene más fases solicitará colocar la cámara en una posición que será donde se ubicará el protagonista al inicio del juego).

Si el objeto colocado sobre el escenario es una puerta, preguntará directamente si se quiere seguir colocando más objetos, por el contrario, si es un ascensor mostrará una serie de ventanas para tener todos los datos necesarios para poder crear el elemento XML correspondiente y en caso de ser un personaje se podrá dar un diálogo.

Si el objeto colocado es un ascensor, la siguiente ventana que mostrará la aplicación será informando que hay que indicar cuáles son las dos plantas que tendrá dicho ascensor, para ello se debe mover el ascensor por el eje Y (subir o bajar) y una vez que llegue a la posición que será una planta “volver a colocarlo”.

Figura 40: Ventana informativa de ascensor

Cuando se hayan indicado las dos plantas del ascensor pedirá la selección del modo de ascensor que tendrá.

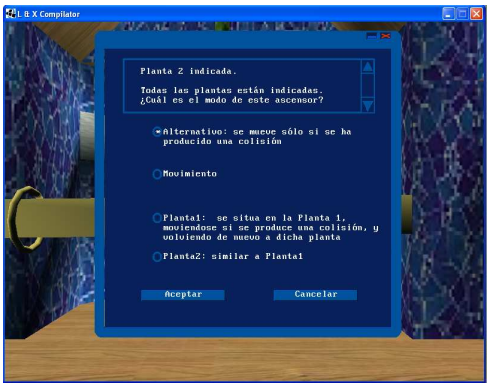

Figura 41: Modo ascensor

Después de indicar el modo de ascensor, pedirá que se elija el tipo de velocidad que tendrá (lento, normal, rápido).

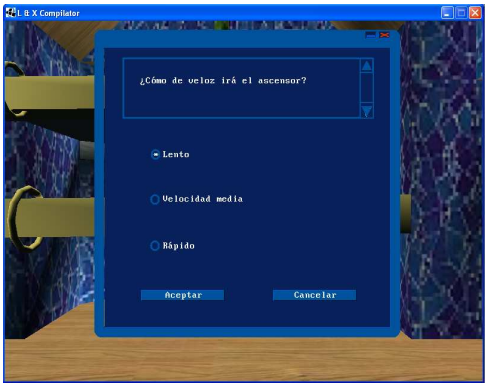

Figura 42: Tipo velocidad

Si se pulsara sobre cancelar en alguna de las ventanas anteriores referentes al ascensor, informaría al desarrollador que en caso de cancelar, el ascensor no sería creado y preguntaría si se quiere cancelar dicha creación. En caso de cancelar, volvería a preguntar si se quieren colocar más objetos en la fase, mientras que por el contrario no se quiere cancelar la creación del ascensor mostraría la ventana anterior que había.

En caso de que el objeto colocado fuera un personaje, la aplicación comprueba si en el directorio correspondiente al archivo .3DS del personaje hay más archivos de este tipo y en caso afirmativo se pregunta al desarrollador si quiere que dicho personaje camine (una evolución de la plataforma inicial), puesto que tiene varios archivos .3DS donde gracias a ellos se puede simular que el personaje anda (ver apartado 3.2.4.).

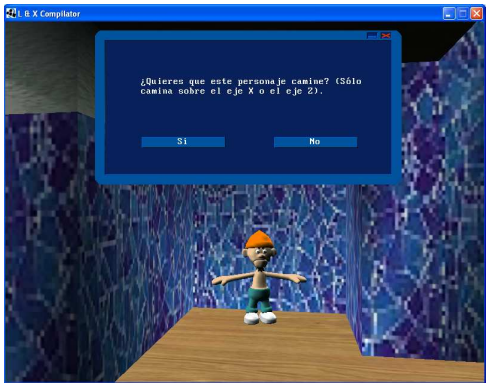

Figura 43: Posibilidad de caminar

Si no se quiere dar movimiento al personaje preguntará si se quiere dar diálogo; por el contrario, si el desarrollador quiere dar movimiento al personaje, la aplicación preguntará sobre el eje que caminará paralelamente el personaje (X o Z) y a continuación se debe indicar el inicio y el final del recorrido (al igual que pasaba en el caso del ascensor con las plantas).

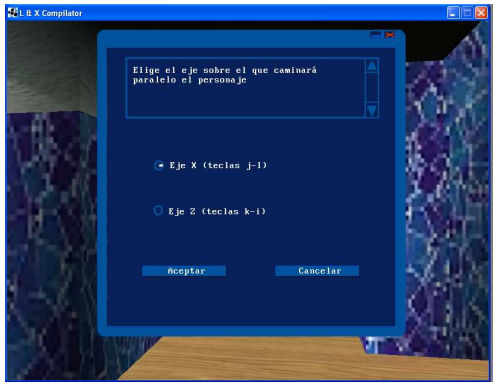

Figura 44: Selección del eje sobre el que caminará el personaje

En el caso de dar a cancelar en la ventana anterior, como pasaba en el caso de ascensores, la aplicación informará que el personaje no será creado y preguntará si se quiere continuar con la cancelación, en ese caso se volverá a preguntar si se quieren crear más objetos dentro de la fase actual.

Una vez indicado el recorrido del personaje, la aplicación preguntará si se quiere dar diálogo al personaje colocado, teniendo la posibilidad de colocar más personajes antes de dar el diálogo y dar diálogo a ese personaje después.

Figura 45: Dar diálogo

Para poder dar el diálogo a un personaje e indicar a qué personaje se le asigna el diálogo se debe colisionar con dicho personaje. Una vez que se colisione con el personaje, la aplicación preguntará por el tipo de intervención que tendrá la conversación (figura 36).

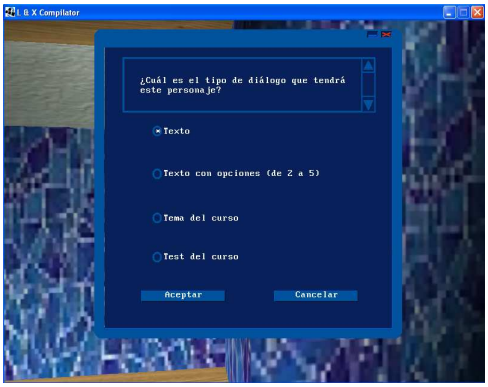

Figura 46: Tipo de intervención

Siempre que se cree una intervención, la aplicación preguntará si se quiere continuar con la conversación del personaje (si es de tipo opción se preguntará continuar por cada una de ellas), en ese caso volverá a mostrar la ventana de selección del tipo de intervención y en caso contrario, preguntará al desarrollador si quiere seguir colocando objetos en dentro de la misma fase.

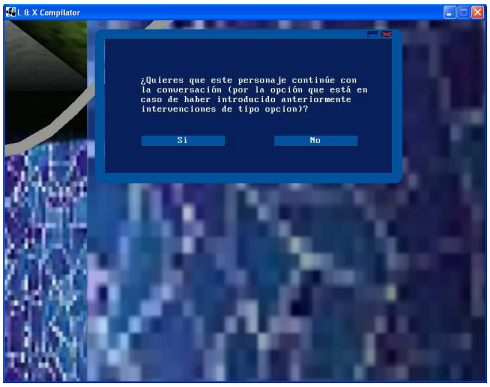

Figura 47: Continuación de la conversación

En el caso de elegir tipo texto, se abrirá una ventana donde se pida al desarrollador que introduzca el contenido de la intervención.

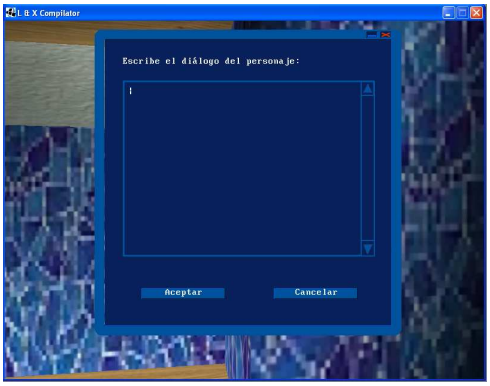

Figura 48: Intervención tipo texto

En el caso de que fuera una intervención de tipo opción, preguntará por el número de opciones (entre 2 y 5) que tendrá la intervención y posteriormente se mostrará una ventana para que se inserte el contenido de la intervención.

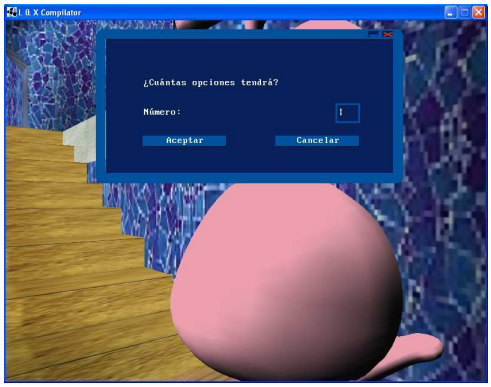

Figura 49: Pregunta por el número de opciones

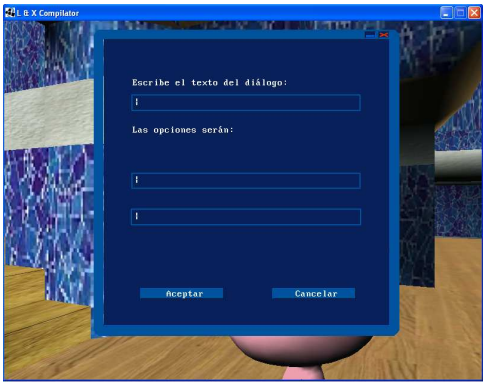

Figura 50: Intervención con 2 opciones

Además, si se quiere continuar con la conversación de esta intervención, la aplicación preguntará si quiere que una opción determinada comparta el resto de conversación con otra o más opciones. Si quiere compartir conversación, pero que no llegue en ese momento, se seguirán creando intervenciones y posteriormente se irá preguntando si después de la última intervención creada llegará la parte común.

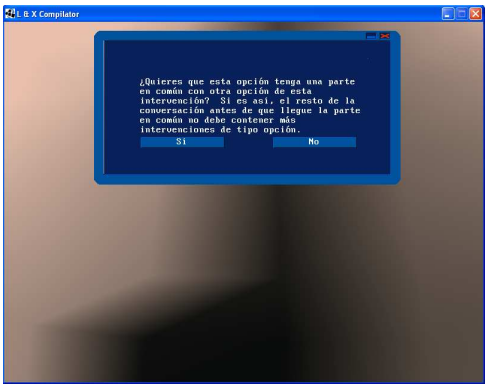

Figura 51: Pregunta si se quiere una parte común

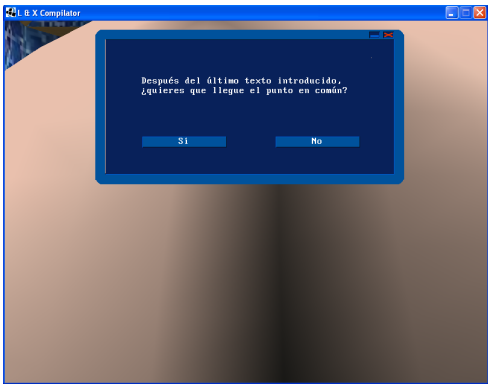

Figura 52: Pregunta si llega la parte común

En caso afirmativo (llega la parte común), si no se ha introducido antes el resto de conversación se crearán las intervenciones compartidas mostrando primero la ventana de selección del tipo de intervención. Por el contrario, si ya hay una conversación compartida se preguntará si se ha introducido anteriormente o si se quiere crear una nueva (en ese caso se volverá a mostrar la ventana de selección del tipo de intervención).

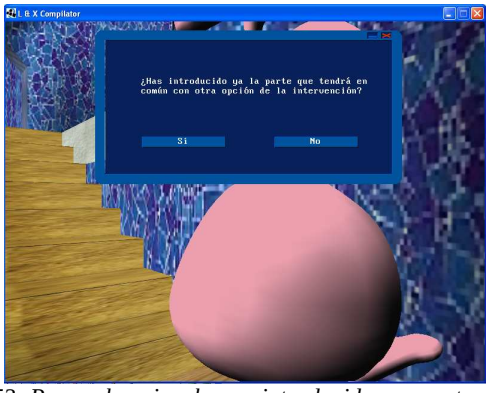

Figura 53: Para saber si ya hemos introducido una parte en común

Si se ha introducido anteriormente, se comprobará si sólo hay una, por lo que se enlazará directamente, pero si hay varias, se mostrará una ventana para que el desarrollador pueda elegir cuál es la conversación compartida que tendrá la última intervención que ha creado.

En el caso de ser una intervención de tipo tema, se asignará el primer tema que no ha sido asignado anteriormente (en alguna intervención de cualquier personaje de que haya en cualquier fase creada).

En el caso de ser una intervención de tipo test, se mostrará una ventana que pedirá el número de preguntas que expondrá la aplicación al usuario final y el número de fallos permitidos para aprobar el examen y así poder cambiar de fase (o lo que el desarrollador quiera).

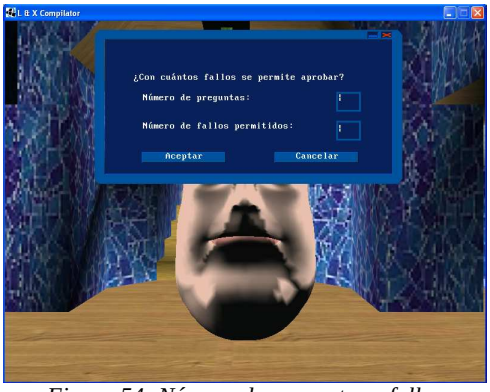

Figura 54: Número de preguntas y fallos

Una vez dados los datos anteriores, la aplicación mostrará una ventana donde el desarrollador pueda elegir el examen que impartirá el personaje.

Figura 55: Selección de tests

Si la aplicación tiene varios archivos de sonido, cada vez que se cree una intervención (excepto de tipo test) se mostrará una ventana para que el desarrollador pueda seleccionar el sonido que puede tener dicha intervención.

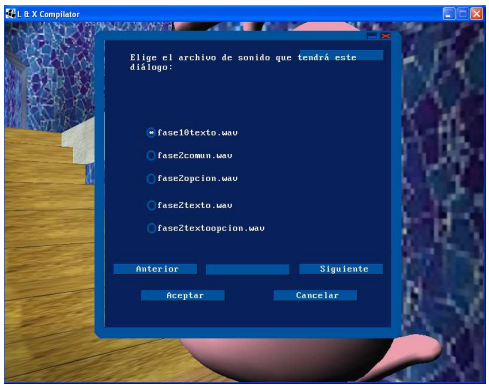

Figura 56: Selección de sonido

Una vez creadas todas las fases que queremos que tenga el juego, la aplicación pedirá enlazar unas con otras (figura 35).

Para ello se situará en la primera fase creada y mostrará una ventana que contenga los diálogos de los personajes (en el caso de que haya varios) para que el desarrollador seleccione uno que pueda llevar a otra fase distinta, si sólo tiene un diálogo, éste será el seleccionado por la aplicación.

Si el diálogo seleccionado no tiene opciones (o no es de tipo test) la siguiente ventana a mostrar es la de selección de fase a la que llevará dicho diálogo.

Por el contrario, si el diálogo seleccionado tiene alguna bifurcación, la aplicación pedirá al desarrollador que seleccione la opción que puede llevar a la siguiente fase.

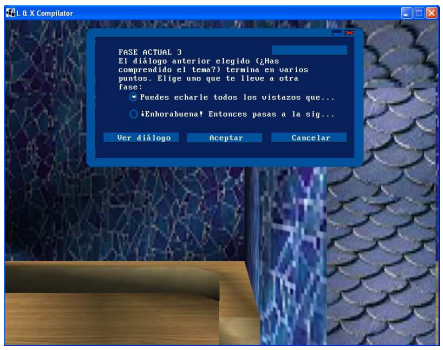

Figura 57: Selección de bifurcación

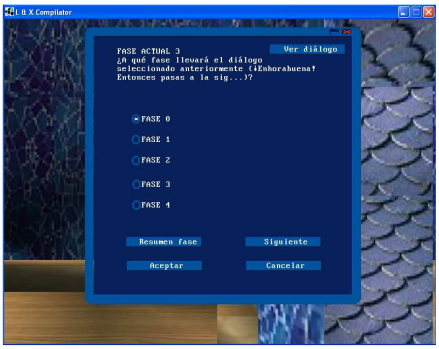

Figura 58: Enlazar fases

Una vez enlazado un diálogo con una fase distinta de la que proviene dicho diálogo, en caso de que esa fase (donde está el diálogo) tenga más diálogos, la aplicación preguntará al desarrollador si quiere seguir eligiendo más diálogos.

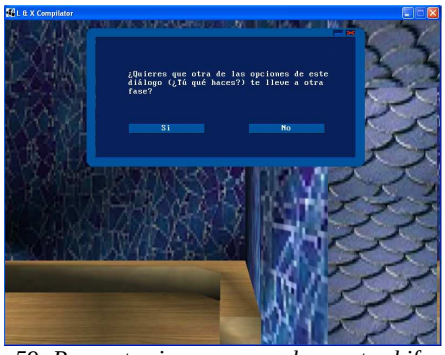

Figura 59: Pregunta si queremos enlazar otra bifurcación

En caso de haber enlazado todos los diálogos de una fase con otras, o no querer seguir enlazando más, la aplicación cambiaría a la siguiente fase creada para mostrar la ventana de selección de diálogos.

Una vez enlazadas las fases, la aplicación pedirá al desarrollador que coloque al protagonista en las coordenadas donde se ubicará al iniciar el juego.

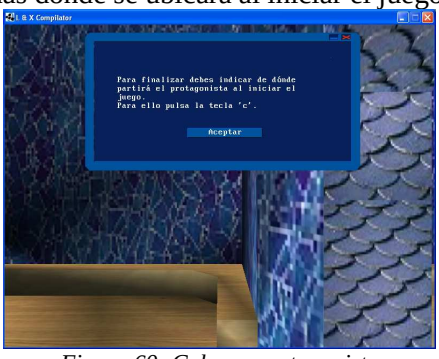

Figura 60: Colocar protagonista

Una vez colocado el protagonista se tiene que dar nombre al fichero para que se genere el guión XML resultado de la edición.

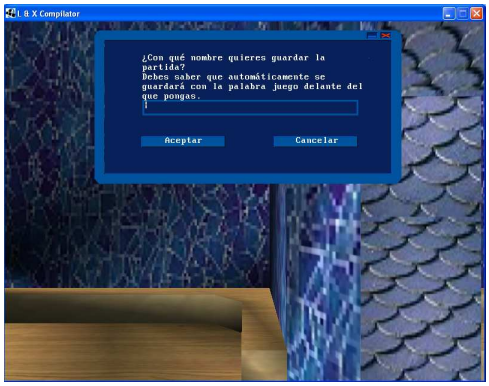

Figura 61: Guardar fichero

Además, cada vez que se cree una fase, se puede ir generando en un archivo los elementos XML correspondientes a lo editado en cada fase creada.

Como consecuencia de que la edición de un juego puede ser larga y por la posibilidad de ir generando el documento XML en la edición de cada fase, la aplicación puede continuar con la edición de nuevas fases a partir de un guión que contiene elementos XML correspondientes a otras fases para continuar con dicha edición y así generar el guión final del juego o ampliar dicho juego. Incluso se pueden añadir fases nuevas a un guión "completo" anteriormente.

3.2.8. Teclado

Para llevar a cabo el manejo del protagonista y/o edición, se creó el objeto teclado, en el cual se registran una serie de teclas para llevar a cabo unas funciones u otras. Las teclas registradas son las siguientes:

- ***Cursores***: Definen el movimiento del protagonista a través del escenario.
- ***ESC***: Evento capturado por el controlador del juego que indica la conclusión de la aplicación.
- ***S***: Tiene dos funcionalidades: 1. Si se está en el modo edición y se presiona esta tecla se decrementará la escala del objeto 3D a colocar en el escenario. 2. Si se juega, al presionarla mostrará un diálogo preguntando si se quiere sonido en el juego.

El resto de teclas sólo son utilizadas en modo edición:

- **A**: Sólo se utiliza su funcionalidad en el modo edición. Al presionar esta tecla aumenta la escala del objeto 3D antes de colocarlo sobre el escenario.
- **PAGE_DOWN**: Con esta tecla se mueve descendentemente por el eje Y el objeto 3D antes de colocarlo sobre el escenario.
- **UP_DOWN**: Con esta tecla se mueve ascendentemente por el eje Y el objeto 3D antes de colocarlo sobre el escenario.
- **G**: Con esta tecla se gira el objeto sobre el eje Y.
- **X**: Con esta tecla se gira el objeto sobre el eje X.
- **Z**: Con esta tecla se gira el objeto sobre el eje Z.
- **C**: Con esta tecla se coloca definitivamente el objeto 3D sobre el escenario.
- **I o K**: Con estas teclas se mueve el objeto 3D sobre el eje Z.
- **L o J:** Con estas teclas se mueve el objeto 3D sobre el eje X.

3.2.9. Ayuda para crear un guión básico

Para finalizar, debido a que la aplicación debe tener un guión mínimo, se comentarán los elementos necesarios para poder crear "a mano" un guión XML elemental coherente con la sintaxis especificada en la DTD 'juegos.dtd'. Este apartado no es una mejora, pero sirve de ayuda para la generación inicial del fichero XML concorde con su DTD.

El resto de elementos que conforman un guión de juegos no será necesaria una explicación detallada puesto que pueden ser generados en el fichero XML por la aplicación de edición.

➢ Definición de temporizadores

Debido a la necesidad de implementar una aplicación escalable, se procedió a realizar un diseño multi-hilo en el que cada elemento susceptible de ser escalado cuenta con un hilo de ejecución.

Debido a ello y a que dichos hilos únicamente precisan ejecutarse cada cierto intervalo de tiempo, aparece el concepto de temporizador, siendo ésta una variable que indica el número de milisegundos entre actuaciones de hilo.

Por lo tanto el hilo está en funcionamiento en intervalos definidos por el temporizador, quedando dormido el tiempo que le sobre después de realizar sus labores. Dicha forma de actuar evita al procesador la carga innecesaria que supondría una espera activa.

Esta tarea no se puede considerar del todo obligatoria en el diseño del juego, puesto que su cometido es proporcionar valores a unos temporizadores que ya cuentan con un valor por defecto. En caso de no asignar valor a estas variables se tomará dicho valor fijo.

Por último, véanse algunas líneas de código XML que implementan tales variables.

```
<Temporizador ascensor="30" dialogo="175" juego="180" personaje="80" protagonista="25" representar="25" teclado="80"/>
```

➢ Definición de los parámetros de pantalla

Antes de empezar con el diseño del juego propiamente dicho, es preciso aportar valor de ciertos parámetros que utilizará la aplicación a la hora de caracterizar el entorno de ejecución. Tales parámetros pertenecen al dominio gráfico de la aplicación y concretan variables tales como el tipo de renderización, el máximo número de polígonos visibles por pantalla, su tamaño, y muchas otras que serán concretadas más adelante.

```
<Pantalla altopantalla="384" anchopantalla="512" collide_offset="250" glcolordepth="16" glfixedblitting="si" glmipmap="si" gltrilinear="si" glzbufferdepth="16" maximo_polys_visibles="30000" redimension="no" renderizacion="hardware" titulo="L &amp;amp; X Compilator" tune_for_outdoor="si"/>
```

➢ Definición de los parámetros del escenario

Apartado del guión encargado de concretar parámetros referentes al entorno virtual en el que se llevará a cabo la aventura. En esta parte de código se hará referencia al fichero utilizado para la renderización del escenario además de otras características como luces del entorno, niebla, y otras muchas variables relacionadas.

```
<Escenario>
  <Mundo activada_caida="si" caida_de_luz="100" cobertura_luz="300">
    <Color_Luz g_luz_ambiente="15" b_luz_ambiente="15" r_luz_ambiente="10"/>
    <Foco_Luz componente_rojo="25" componente_verde="25" componente_azul="25" posicion_x="850" posicion_y="-150" posicion_z="-430"/>
          ……
  </Mundo>
  <Entorno directorio_texturas="juegos-Geografia-3ds-escenario" archivo_3ds="juegos-Geografia-3ds-escenario-ql.3ds" escala="20f"/>
</Escenario>
```

➢ Definición de parámetros del protagonista

Sección del guión del juego que hace referencia a la caracterización del protagonista en la que se da valor a variables tales como su altura, envergadura, velocidad, posición inicial, y otras muchas variables.

```
<Protagonista posicion_x="620" posicion_y="-60" posicion_z="-600" grosor="6" altura="30" velocidad_caida="4f" velocidad_movimiento="2.5f" velocidad_giro="0.05f" rotacion_y="3.14f"/>
```

4.- Implementación

4.1.- Núcleo de la aplicación

En este apartado se analizará la implementación de la aplicación, se expondrá la estructura de clases, así como las funciones que desempeñan cada una de ellas. Se pasará a caracterizar las partes integrantes, concretando la finalidad de cada una y los medios que utiliza para realizar su cometido.

4.1.1. Motivación del diseño

La estructura de la aplicación se encuentra descompuesta en diversas clases, cada una de ellas de una índole diferente. Esto es debido a las diferentes áreas de trabajo del proyecto y a la necesidad de separar cada una ellas. Estas áreas son: entorno gráfico, entorno de diálogo, entorno de ventanas, entorno de control, entorno docente y generación de guiones.

El entorno de gráfico engloba todas las clases involucradas en la definición de la parte gráfica que caracteriza a la plataforma. Todas estas entidades harán uso de las herramientas que aporta el motor gráfico jPCT y tienen como finalidad la composición íntegra de la escena en la que se desenvolverá el personaje.

El entorno de diálogo está formado por las clases encargadas de la construcción de todos y cada uno de los diálogos que aparecerán en la aventura. Estas clases implementan tanto la estructura, contenido y métodos de funcionamiento, capaces de sintetizar por completo las ventanas de diálogo utilizadas para conversar con un personaje.

El entorno de ventanas está formado por todas las clases encargadas de la construcción de una ventana gráfica que contiene texto y se muestra dentro de la aplicación.

El entorno de control está formado por las clases necesarias para controlar el estado del juego y las transiciones de fase producidas en éste.

El entorno docente está formado por las clases necesarias para generar los ficheros XML docentes (curso y tests). Este entorno no utiliza el motor JPCT, y se invoca en el modo de edición para generar dichos ficheros.

Por último, la generación de guiones está formada por las clases necesarias para crear ficheros XML que permitan jugar en la aplicación. Estos ficheros pueden ser creados desde la parte edición si se está generando el fichero desde cero, o por el contrario desde la parte juego si se quiere guardar una partida.

Las siguientes figuras presentan el diagrama de clases divididos en los módulos mencionados anteriormente.

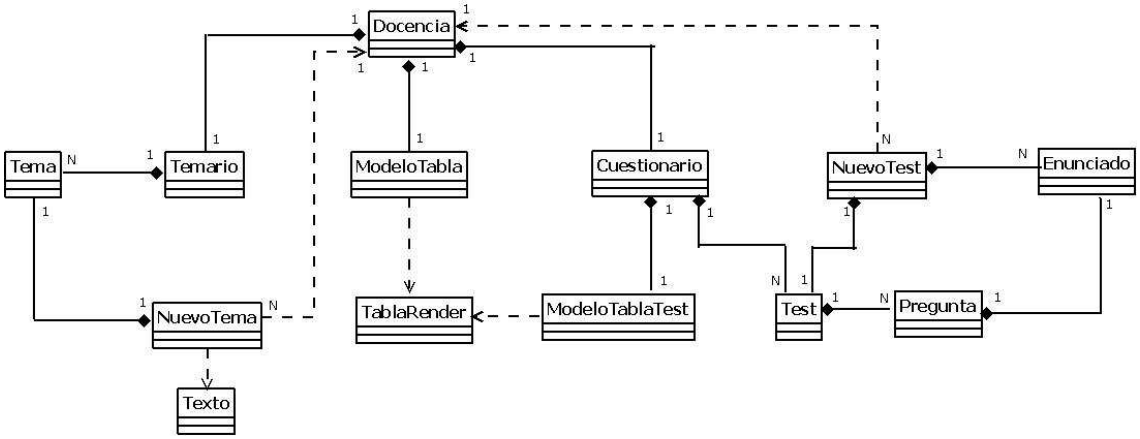

Figura 62: Diagrama de clases dedicadas a la docencia

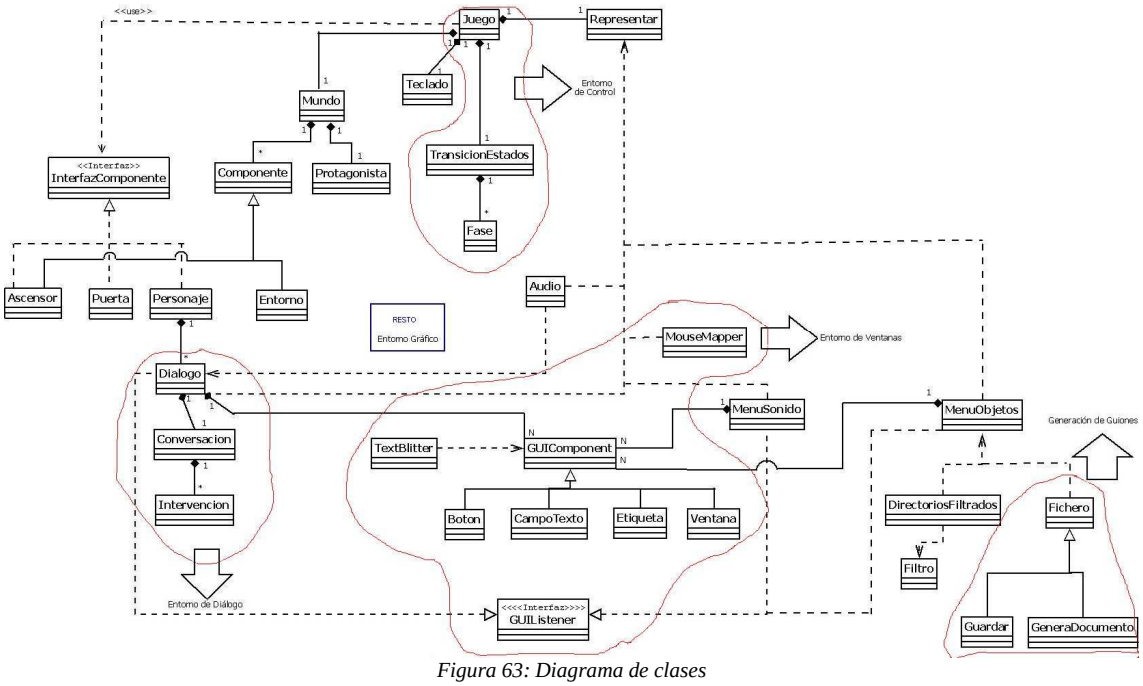

Figura 63: Diagrama de clases

Nota:

En los siguientes apartados se ha pretendido ofrecer unas pinceladas del núcleo de la aplicación, todo ello trabajado de forma sumamente concisa y sin entrar en ningún detalle de bajo nivel.

Sería preciso completar los conceptos aquí expuestos con la documentación de la aplicación. En ella se tratan estos mismos temas de forma más concreta, ofreciendo una visión de bajo nivel del núcleo de la aplicación y todas las operaciones que en él tienen cabida. Para tal fin, y debido a la gran extensión de tal documentación, ha quedado adjunta al final del proyecto (excepto las clases que no han sido modificadas en esta versión y las menos relevantes). (ver anexo IV)

4.1.1.1. Entorno gráfico

Está constituido por aquellas clases que hacen posible la construcción de la escena y todo lo que acontece en ella (figura 63).

Debido a esto el ámbito gráfico de la aplicación está definido por clases que representan cada uno de los elementos empleados en la construcción del escenario, entre los que se encuentran el protagonista, los personajes, los objetos inertes (puerta), los ascensores, el modelo que define el entorno virtual donde se desenvuelve la historia, el controlador del teclado y el controlador de la representación. Muchos de ellos contarán con un hilo de ejecución paralelo al hilo principal que permitan ejecutar cíclicamente las rutinas que implementan su funcionamiento.

Las entidades que componen el aspecto gráfico de la aplicación van a ser las siguientes:

- *Clase 'Componente'*

Se puede definir como la clase genérica de la que heredan todos los componentes visibles de la escena, por ello será la encargada de la implementación de los atributos y métodos compartidos por todos.

Se trata de una clase abstracta puesto que consta de elementos sin implementar encargados de la particularización de alguna de las funciones que implementa. Así el elemento en cuestión que herede de esta clase será el encargado de implementar dicho comportamiento particular.

Las clases que heredaran de esta abstracción son: 'Ascensor', 'Entorno', 'Personaje', 'Puerta' y 'MenuObjetos' (ésta última en caso de editar el guión).

Gracias a la recodificación de esta clase, se pueden girar los objetos sobre su eje (indicando por parámetro el ángulo de giro), colocar dichos objetos en el mundo indicando su posición… para poder tenerlos en el escenario tal y como el desarrollador los ha ubicado, por lo que se

puede decir que esta clase ha sido modificada para poder llevar a cabo la segunda versión de la aplicación.

Para finalizar es importante destacar el atributo de tipo 'Object3D', encargado de almacenar el objeto tridimensional característico de dicho componente y que sirve de enlace con el motor jPCT.

```
public abstract class Componente {
     public boolean activado=false;
     public int id_componente=-1;
     public Object3D objeto=null;
     public float rotfija=0;
     public SimpleVector posfija=new SimpleVector (0,0,0);
     private float angulo=0f;
     private float anguloX=0f;
     private float anguloZ=0;
     private String archivo3D="";
     private String directorioTexturas="";
     private float escala;

     public Componente (){
         objeto=new Object3D(0);
     }
     public Componente (Object3D obj){
         objeto=new Object3D(obj);
     }
     public Object3D getObjeto(){
        return objeto;
     }
     public void reposicionarComponente (SimpleVector pos_ini, float rot_ini){  ....   }
     public TextureManager recogerTexturas(String ptexturas){   ....     }
     public void constructorComponente (String path3ds, float escala){
         ....
         perfilarComponente();
     }
     public Object3D constructorComponente(String path3ds){   ....    }
     public boolean getActivacion(){
        return activado;
     }
     public int getIdComponente(){
         return id_componente;
    }
     public void rotarX(float angulo){
         anguloX=angulo;
         objeto.rotateX(angulo);
    }
    public void rotarZ(float angulo){
         anguloZ=angulo;
         objeto.rotateZ(angulo);
    }
    public void setOrigen(SimpleVector origen){
         posfija=origen;
    }
    public SimpleVector getOrigen(){
         return posfija;
    }
       ....
    abstract public void perfilarComponente();
}
```

A diferencia de la primera versión, se han añadido nuevos atributos: `angulo`, `anguloX`, `anguloZ`, `archivo3D`, `directorioTexturas` y `escala` (con sus correspondientes métodos get/set) para facilitar la edición del juego. Así cuando se quiera colocar un objeto 3D en el escenario estos atributos almacenarán los valores que utilizan dichos objetos 3D.

Otro método nuevo ha sido `constructorComponente(String path3ds)`, que hace prácticamente la misma función que el de `constructorComponente(String path3ds, flota escala)` con la diferencia clara de imponer una escala fija para todos los objetos y además no se invoca al método `perfilarComponente`, puesto que se utiliza en la edición y todavía no hemos colocado fijamente dicho objeto.

- *Interfaz 'InterfazComponente'*

Interfaz que define el conjunto de métodos que dan lugar a la API que utilizarán clases externas para acceder al comportamiento de los objetos 'Ascensor', 'Personaje' y 'Puerta'.

```
public interface InterfazComponente{
    public void posicionarComponente (SimpleVector posicion_ini, float rotacion_ini);
    public void activarComponente();
    public void desactivarComponente();
}
```

Contiene los métodos encargados de llevar a cabo procesos de activación, desactivación y posicionamiento.

Esta clase se mantiene igual que en la primera versión, no ha sido modificada.

- *Clase 'Mundo'*

Extiende de la clase 'World' del motor jPCT, por lo que heredará todas sus características y funcionalidad.

Se trata del objeto principal de todo juego creado con la plataforma, constituyendo el entorno donde se carga la escena completa. Una aplicación sin el objeto 'Mundo' no tendría ningún sentido ya que no existiría la base necesaria para asentar todos los objetos que forman parte del juego. En resumen, se puede definir este objeto como el mundo donde se desenvolverá toda la aventura gráfica.

Inicialmente parte como un ente vacío al que habrá que asociar objetos tales como el entorno, los personajes, y todo aquello que forme parte de la aventura.

El constructor se encargará de realizar una caracterización inicial en lo que respecta a las propiedades de iluminación y los métodos definidos aportarán las herramientas para la incorporación de fuentes lumínicas al objeto creado.

```
    public class Mundo extends World {
       private SimpleVector posicion;
       public Mundo (boolean caidaluz,float caida,float cobertura,int rojoambiente, int
verdeambiente, int azulambiente){
         super();
         ....
         Config.fadeoutLight=caidaluz;
         Config.tuneForOutdoor();
         if (caidaluz){
            Config.linearDiv=caida;
            Config.lightDiscardDistance=cobertura;
         }
         setAmbientLight(rojoambiente,verdeambiente,azulambiente);
      }
       public boolean anadirluces (SimpleVector[] posluces,Color[] colorluces){   ....   }
       public void setPosicion(SimpleVector posicion){
         this.posicion=posicion;
      }
       public SimpleVector getPosicion(){
         return posicion;
      }
   }
```

Esta clase ha sido modificada en la versión 2. Por ello se han añadido el atributo posición y los métodos `getPosicion()` y `setPosicion(…)`. El atributo es la variable que almacena la posición de la cámara (modificada una pequeña distancia) para poder mostrar un objeto que queremos colocar en el mundo cerca de la cámara, esto es así para facilitar la visión del nuevo objeto 3D que no está posicionado en nuestra aplicación (la posición que tiene se puede modificar).

- *Clase 'Ascensor'*

Extiende de la clase 'Componente' e implementa los interfaces 'Runnable' e 'InterfazComponente'. Cada objeto creado representará un ascensor que se ubicará dentro del escenario.

```
    public class Ascensor extends Componente implements InterfazComponente,Runnable {
      ....
      public static int MUESTREO=30;
      ....
      public boolean vivo;
      private int modo=Ascensor.MODOALTERNATIVO;
```

```
      ....
       public Ascensor(int idasc,String a3ds,String atexturas,float escala,float plantauno, float
plantados, float velocid){
         super();
         vivo=true;
         hiloact=null;
         id_componente=idasc;
         // Recoge las texturas del directorio indicado y llama a construir elemento.
         manejador_texturas=recogerTexturas(atexturas);
         constructorComponente(a3ds,escala);
         PLANTA1=plantauno;
         PLANTA2=plantados;
         VEL_ASC=velocid;
      }
      public void moverAscensor(){
         float auxmov,auxres,paso;
         if (temporiz>0) temporiz--;
         else{
            // Calcula el proximo movimiento.
            auxmov=posicion+(proxmovimiento*VEL_ASC);
           auxres=0;
            if (PLANTA1>PLANTA2){
               if (auxmov<PLANTA2)auxres=(PLANTA2-auxmov);
               else if (auxmov>PLANTA1)auxres=(auxmov-PLANTA1);
            }
            else{
               if (auxmov>PLANTA2)auxres=(auxmov-PLANTA2);
               else if (auxmov<PLANTA1)auxres=(PLANTA1-auxmov);
            }
            paso=(VEL_ASC-auxres)*proxmovimiento;
            posicion+=paso;
            objeto.translate(0,paso,0);
            if (posicion==PLANTA1 || posicion==PLANTA2){
               temporiz=40;
               proxmovimiento*=-1f;
            }
         }
      }
      ....
   }
```

Su creación pasa por la carga del modelo y texturas que definen su estética, además de la asignación de parámetros tales como tamaño, posición, trayecto que realiza, y modo de funcionamiento. Esta última variable se encuentra relacionada con los diversos tipos de movimientos implementados, que permiten ajustar el funcionamiento del componente lo máximo posible al escenario que lo engloba.

Todo objeto de esta clase se ejecuta desde un hilo diferente, encargado de realizar la comprobación de colisiones y movimiento del objeto tridimensional reflejado en la escena. Su ciclo de vida viene caracterizado por un bucle que terminará cuando el controlador del juego decida cerrar la aplicación. Este bucle únicamente se preocupará de las labores que conciernen a este objeto (para mover el ascensor), las cuales se repetirán periódicamente cada cierto tiempo controlado por un temporizador, de modo que en el tiempo restante el hilo permanecerá dormido (de esta forma se evita añadir carga innecesaria al procesador).

```
       public void run (){
         ....
         desactivarComponente();
         while(vivo){
            tempor_actual=System.currentTimeMillis();
            switch(modo){
               case (MODOALTERNATIVO):
                  if((activado && objeto.wasTargetOfLastCollision())||(posicion!=PLANTA1 &&
posicion!=PLANTA2))
                     moverAscensor();
                  break;
               case (MODOMOVIMIENTO):
                  if(activado)
                     moverAscensor();
                  break;
               case (MODOPLANTA1):
                  if((activado && objeto.wasTargetOfLastCollision()) ||(posicion!=PLANTA1))
                     moverAscensor();
                  break;
               case (MODOPLANTA2):
                  if((activado && objeto.wasTargetOfLastCollision()) ||(posicion!=PLANTA2))
                     moverAscensor();
                  break;
            }
```

```
            // Una vez hecho el movimiento se espera un tiempo que viene indicado por la variable
MUESTREO
            objeto.resetCollisionStatus();
            try {
               tempor2=System.currentTimeMillis();
               if ((tempor2-tempor_actual)<MUESTREO)
                  Thread.sleep(MUESTREO-tempor2+tempor_actual);
            }
                catch (Exception e){
               }
         }
      }
```

Esta clase sigue igual que en la primera versión.

- *Clase 'Entorno'*

Toda aplicación diseñada con la plataforma ha de contar con un objeto del tipo 'Entorno'. Esta clase implementa los atributos y métodos para trabajar con dicha entidad.

Hereda de 'Componente', e implementa los métodos abstractos definidos en esa clase, consiguiendo particularizar el funcionamiento de algunos de ellos.

```
    public class Entorno extends Componente {
      ....
      public Entorno (String p3ds, String ptexturas, float escala){
         super ();
         manejador_texturas=recogerTexturas(ptexturas);
         constructorComponente(p3ds,escala);   ___
      }
      public void perfilarComponente(){
         OcTree arbol=new OcTree (objeto, 100, OcTree.MODE_OPTIMIZED);
         arbol.setCollisionUse(OcTree.COLLISION_USE);
         objeto.setOcTree(arbol);
         objeto.enableLazyTransformations();
      }
   }
```

El objeto entorno será el encargado de almacenar el escenario donde se desenvuelve el juego así como los métodos utilizados para su construcción y atributos que lo caracterizan. Por su parte el constructor se encargará de la inicialización de tal escenario, procedimiento que complementará con la carga del modelo 3DS, texturas y dimensión que lo definen, atributos que serán pasados como parámetro en la creación del objeto.

Finalizada la creación del objeto entorno, se pasará a añadírsele al objeto 'Mundo' creado previamente. La entidad 'Entorno' se caracteriza por ser meramente pasiva, no añadiendo ningún tipo de funcionalidad posterior a su creación.

Se puede decir que la clase no ha sufrido modificación alguna con respecto a la segunda versión.

- *Clase 'Personaje'*

Los objetos pertenecientes a esta clase representarán a cada uno de los personajes involucrados en la historia. El constructor inicializará el objeto correspondiente cargando el modelo y texturas que lo caracterizan. Es importante destacar que esta clase hereda de 'Componente' y, a igual que las mencionadas hasta ahora, implementa sus métodos abstractos particularizando así el comportamiento de algunas de las funciones definida por la clase abstracta. También implementa la interfaz 'InterfazComponente' definiendo cada uno de sus métodos.

```
   public class Personaje extends Componente implements InterfazComponente, Runnable{
      ....
      private MenuObjetos menu;
      private float posfinal;
      private float ind=0.1f;
      private boolean edicion;
      private boolean andar;
      private String eje="y";
```

```
      private Representar representar;
      public Personaje(int idper, String p3ds, String ptexturas, float escala){
          super();
          constructor(idper, p3ds, ptexturas, escala);
          constructorComponente(p3ds,escala);
      }
      public Personaje(int idper, String p3ds, String ptexturas, float escala, String eje){
         super();
         constructor(idper, p3ds, ptexturas, escala);
         this.eje=eje;
         andar=true;
         int ind1=p3ds.lastIndexOf(File.separatorChar);
         String directorio=p3ds.substring(0,ind1);
         constructorComponente(p3ds,escala);
         crearAnimacion(directorio, escala);
       }
       private void constructor(int idper, String p3ds, String ptexturas, float escala){
         vivo=true;
         hiloact=null;
         dialogo=null;
         id_componente=idper;
         hilodialogo=null;
         setDirectorioTexturas(ptexturas);
         setArchivo3D(p3ds);
         manejador_texturas=recogerTexturas(ptexturas);
      }
  ....
}
```

Al igual que el objeto ‘Ascensor’, un objeto ‘Personaje’ implementa la clase ‘Runnable’. Esto indica que correrá en un hilo paralelo al principal de modo que su método ‘run’ defina el comportamiento de dicha línea de ejecución.

Este método está basado en un bucle, activo hasta el momento que el hilo principal cierre la aplicación por completo, en el que se van a realizar las siguientes operaciones:

- Movimiento del personaje en el caso de ser necesario. En esta versión se pueden diferenciar dos tipos de movimientos: sobre el eje Y y/o caminando. En el primer caso sólo hay que dar valor al atributo movimiento del elemento XML Personaje del guión del juego y así dicho personaje realizará pequeños giros sobre su eje Y, mientras que para caminar hay que indicar sobre qué eje caminará (X o Z) e indicar la posición final del recorrido (la distancia desde la posición inicial, indicada ésta por las coordenadas posición_x, posición_y, posición_z), y en este caso si el personaje llega al final del recorrido realiza un giro de 180º y hace el camino de vuelta, así sucesivamente.

```
   private void moverPersonaje(){
      float rotahora=direccionrot*velocidadrotacion;
      rotacion_acumulada=rotacion_acumulada+rotahora;
      objeto.rotateY(rotahora);
      if (rotacion_acumulada>=(0.5f)||rotacion_acumulada<=(-0.5f))
         direccionrot=direccionrot*(-1);
   }
   private void andar(int x){
      ind+=0.15f;
      if (ind>=1f)
         ind=0f;
      objeto.animate(ind);
      SimpleVector posicion=objeto.getTranslation();
      float posX=posicion.x;
      float posZ=posicion.z;
      if(eje.equals("x")){//si andamos en el eje X
         if(posX==posfinal){
            objeto.rotateY((float)-Math.PI);//giramos el objeto 180º
         }
         else if(posX==0){
           objeto.rotateY((float)-Math.PI);//giramos el personaje 180º
         }
         objeto.translate(x,0,0);
      }
      else if(eje.equals("z")){//si andamos en el eje Z
          if(posZ==posfinal){
             objeto.rotateY((float)-Math.PI);
          }
          else if(posZ==0){
             objeto.rotateY((float)-Math.PI);
          }
          objeto.translate(0,0,x);
```

```
    }
}
```

Cabe destacar, que para que un personaje pueda caminar antes se deben haber obtenido los fotogramas que simulen dicha caminata (si se trata de un objeto 3D), puesto que en caso contrario no se puede crear la animación (y en caso de "obligar" a caminar, el objeto se moverá sobre el eje indicado pero sin simularlo, es decir a pesar de recorrer el eje parecerá "estático" sin mover las piernas):

```
    private void crearAnimacion(String directorio, float escala){
       DirectoriosFiltrados df=new DirectoriosFiltrados();
       Vector objetos3D=df.listarDirectoriosFiltro(directorio, ".3ds");
       if(objetos3D.size()>1){
          Animation animacion=new Animation(objetos3D.size());
          animacion.createSubSequence("andar");
          for(int i=0; i<objetos3D.size(); i++){
             File archivo=(File)objetos3D.elementAt(i);
             String nombre=archivo.getAbsolutePath();
             constructorComponente(nombre, escala);
             animacion.addKeyFrame(objeto.getMesh());
           }
           objeto.setAnimationSequence(animacion);
       }
       else
           setEje("y");
    }
```

- Comprobar si se produjo alguna colisión, y en caso afirmativo:
    - Comprobar si se está en modo edición. En ese caso se mostrará una serie de ventanas para que se dé el diálogo correspondiente al personaje si éste no tiene asignado ya un diálogo.
    - En caso de no estar en modo edición, se comprueba si dicho personaje se encuentra caracterizado por un diálogo en la fase actual. Si así es, lanzar un nuevo hilo que ejecute el diálogo correspondiente.
    - Finalizar el diálogo y pasar el control al controlador del juego. Si el diálogo del que se proviene ha modificado la fase en la que se encuentra el usuario, el controlador será el encargado de hacer realidad estos cambios.
    - Proceder a realizar una espera de 5 segundos en los que no se registrará ningún tipo de colisión con el personaje.

```
    public void run (){
       if(!edicion)
         desactivarComponente();
       long tempor_actual;
       long tempor2;
       long cincosegprim,cincosegseg;
       boolean borrar_dialogo=false;
       cincosegprim=System.currentTimeMillis();
       int x=0;
       SimpleVector posicion=objeto.getTranslation();
       float posX=posicion.x;
       float posZ=posicion.z;
       if(eje.equals("z")){
          if(posfinal>0)
             x=2;
          else
             x=-2;
       }
       else{
          if(posfinal>0)
             x=-1;
          else
             x=1;
       }
       while(vivo){
          tempor_actual=System.currentTimeMillis();
          if (activado && activado_movimiento) moverPersonaje();
          if(!edicion){
             if(activado && andar){
                posicion=objeto.getTranslation();
                posX=posicion.x;
                posZ=posicion.z;
```

```
                    if(posX==posfinal || posX==0)
                       x=-x;
                    andar(x);
                }
                if (activado && (dialogo!=null)){
                    dialogo.setRepresentar(representar);
                    cincosegseg=System.currentTimeMillis();
                    if (cincosegseg-cincosegprim>TIEMPOENTRE){
                       if(objeto.wasTargetOfLastCollision()){
                          while(!dialogo.maquinaestados.cerrarCandado(true)){}
                          hilodialogo=new Thread (dialogo);
                          dialogo.maquinaestados.TPsetState(false);
                          dialogo.maquinaestados.setDD(true);
                          hilodialogo.start();
                          try{
                             hilodialogo.join();
                          }
                              catch (Exception e){
                                System.out.println("Se produjo un error al esperar a que se cerrase la
ventana.");
                                System.out.println("Se procederá a cerrar la ventana.");
                                borrar_dialogo=true;
                                dialogo.ind_preg=0;
                                dialogo.nota=0;
                                dialogo.maquinaestados.estado_siguiente=-1;
                                dialogo.conversacion.cerrarConversacion();
                                dialogo.vivo=false;
                             }
                          objeto.resetCollisionStatus();
                          dialogo.maquinaestados.abrirCandado();
                          if (borrar_dialogo){
                             dialogo=null;
                             borrar_dialogo=false;
                          }
                          cincosegprim=System.currentTimeMillis();
                       }
                    }
                 }
              }
              else{
                 if (activado){
                    cincosegseg=System.currentTimeMillis();
                    if (cincosegseg-cincosegprim>TIEMPOENTRE){
                       if(objeto.wasTargetOfLastCollision()){
                          if(menu!=null){
                             menu.setIdPersonaje(id_componente);
                             menu.tipoDialogo("");
                             boolean espera=menu.esperar();
                             while(espera){
                                espera=menu.esperar();
                             }
                          }
                          objeto.resetCollisionStatus();
                          cincosegprim=System.currentTimeMillis();
                       }
                    }
                 }
              }
              try{
                 tempor2=System.currentTimeMillis();
                 if ((tempor2-tempor_actual)<MUESTREO)
                    Thread.sleep(MUESTREO-tempor2+tempor_actual);
              }
                  catch (Exception e){}
           }

        }
```

Si se está jugando, el cambio de fase que realiza el controlador del juego se encargará de definir cuál de estos personajes se encuentra activo y cuál no, asignando a los activos el diálogo correspondiente.

Si un personaje no se encuentra activo en una determinada fase, se procede a dormir el hilo que lleva a cabo su ejecución. En caso contrario se reanuda la actividad del hilo. Con esta técnica se consigue que el procesador no gaste recursos innecesarios.

```
      public void setDialogo(Dialogo dial){
         dialogo=dial;
      }
      public void setThread(Thread tt){
         hiloact=tt;
      }
      public void activarComponente(){
         activado=true;
```

```
        ....
        hiloact.resume();
    }
    public void desactivarComponente(){
        activado=false;
        ....
        hiloact.suspend();
    }
```

Otra decisión de diseño se refiere a los tiempos de espera en los que se ve envuelto cada hilo, una vez que se termina la ejecución de las rutinas pertenecientes a un periodo y no habiendo agotado el tiempo que se le concedió. En estos casos, se rechazó la espera activa como opción, optándose por dormir el hilo este tiempo sobrante.

Para finalizar, cabe decir que esta clase ha sido modificada para poder llevar a cabo el recorrido y animación de un personaje.

- *Clase 'Puerta'*

Los objetos de esta clase constituyen todos los objetos inertes que aparecen en la escena. Se le dio este nombre tan confuso debido a que con él se definieron las puertas que impedían el paso a otras zonas de la escena. A medida que avanzó el diseño, el concepto se extendió, pasando a ser la entidad bajo la que se engloba cada uno de los elementos que existentes en el entorno sin ninguna otra función mas que la de estar ahí.

La clase hereda de 'componente' y, a igual que sus precedentes, el constructor será el encargado de dar valor al atributo 'objeto' de la clase padre por medio de la carga del modelo 3DS y texturas que definen la forma del mismo.

También lleva a cabo la implementación de la interfaz 'InterfazComponente', dando funcionalidad así a las rutinas encargadas del posicionamiento, así como habilitar y deshabilitar los elementos renderizados en la escena.

```
 public class Puerta extends Componente implements InterfazComponente{
  ....
  public Puerta (int idpuer, String puer3ds, String puertexturas, float escala){
     super();
     id_componente=idpuer;
     manejador_texturas=recogerTexturas(puertexturas);
     constructorComponente(puer3ds,escala);
  }
   public void perfilarComponente(){   ....     }
  public void posicionarComponente (SimpleVector posicion_ini, float rotacion_ini){  ....   }
  public void activarComponente(){
     objeto.enableCollisionListeners();
     objeto.setVisibility(Object3D.OBJ_VISIBLE);
     activado=true;
  }
  public void desactivarComponente(){
     objeto.disableCollisionListeners();
     objeto.setVisibility(Object3D.OBJ_INVISIBLE);
     activado=false;
  }
}
```

Clase sin cambio alguno en la segunda versión.

- *Clase 'Protagonista'*

Toda aplicación construida con la plataforma tiene que tener un objeto de esta clase encargado de la representación del usuario del juego en el entorno virtual creado. Esta entidad define los métodos y atributos que definen el comportamiento y características del protagonista. Así por ejemplo, dentro de estas definiciones se encuentran la caracterización de su volumen, su posición, rutinas que implementan los movimientos que puede realizar, etc.

```
public class Protagonista implements Runnable {
    ....
    public static int MUESTREO = 30;
    ....
```

```
       public Protagonista(World mundo, SimpleVector posini, float alt, float rad, float caida, float
movim, float giro){
         this.mundo=mundo;
         vivo=true;
         ojo=mundo.getCamera();
         POSICION_INICIAL=posini;
         ALTURA=alt;
         RADIO_ESFERA_COLISION=rad;
         VEL_CAIDA=caida;
         VEL_MOV=movim;
         VEL_GIR=giro;
         // Creamos la elipse que conforma el volumen del protagonista y ubicamos a este en la
posicion adecuada
         CAMPO_PROTECTOR=new SimpleVector (RADIO_ESFERA_COLISION,ALTURA/2f,RADIO_ESFERA_COLISION);
         ojo.setPosition(POSICION_INICIAL);
      }
```

Igual que en casos anteriores, esta clase implementa la interfaz 'Runnable', lo que implica que su ejecución se hará en un hilo paralelo al que ejecuta el control del juego. Este hilo de nuevo se caracteriza por estar continuamente ejecutando un bucle encargado de llevar a cabo determinadas tareas cada cierto tiempo, permaneciendo dormido el tiempo sobrante. Entre otras, las tareas que se repiten periódicamente, son:

- Comprobar si se produjo un evento nuevo al pulsar una tecla. Para ello se accederá al controlador del teclado implementado por el objeto de la clase 'Teclado'.
- En caso afirmativo realizar el movimiento correspondiente con el que se mueve la cámara/protagonista.

```
     private void moverProtagonista(){
         SimpleVector posahora;
         movCaer();
         if (teclado.getAdelante()) movAdelante();
         if (teclado.getAtras()) movAtras();
         ....
      }
      public void run (){
         ....
         while(vivo){
            tempor_actual=System.currentTimeMillis();
            if(!edicion){
               if (transic.TPgetState())
                  moverProtagonista();
            }
            else
               moverProtagonista();
            try{
               tempor2=System.currentTimeMillis();
               if ((tempor2-tempor_actual)<MUESTREO)
                  Thread.sleep(MUESTREO-tempor2+tempor_actual);
            }
                catch (Exception e){}
         }
      }
```

Llegados a este punto, es preciso comentar la necesidad de caracterizar al protagonista con un volumen determinado que permita implementar una detección de colisiones basadas en elipsoides.

Todo movimiento realizado tendrá en cuenta dicho volumen a la hora de llevarse a cabo de modo que, en caso de producirse alguna colisión con cualquier elemento de la escena (ya sea pared, personaje o puerta), se llevará a cabo una obstaculización de dicho desplazamiento.

Clase modificada por la segunda versión para poder mover la cámara sin ningún problema y no sufrir un "bloqueo" ante un posible cierre de candado.

- *Clase 'Representar'*

Toda aplicación consta de un objeto de tipo 'Representar'. Éste será el encargado de llevar a cabo la representación en pantalla de la sucesión de imágenes renderizadas que constituyen el juego, así como la visualización de las diversas ventanas/diálogos (por lo que comprobará eventos producidos por el ratón y/o teclado en caso de estar metiendo datos en la aplicación).

Dependiendo del tipo de renderización que se haya especificado en el archivo XML se procederán a ejecutarse ciertas rutinas correspondientes a dicho modo.

Esta clase ha sufrido modificaciones para poder llevar a cabo la edición del guión de juego y mostrar las ventanas dentro de la aplicación.

```
    public class Representar implements Runnable {
        public static int MUESTREO=25;
        ....
        // Indica si la rederización se realizará por medio de Hardware o software
        private boolean software;
        private MenuObjetos menu;
       private MouseMapper mouse;
       private KeyMapper keyMapper;
        private Dialogo dialogo;
        private Audio audio;
        private MenuSonido sonido;
        ....
        public Representar(Mundo mundo, int ancho, int alto,Graphics gFrame, Frame frame,boolean
software,TransicionEstados transic){
         ....
         vivo=true;
         ....
         // Inicialmente se dispone la renderización software.
         imagenes.enableRenderer(IRenderer.RENDERER_SOFTWARE);
         ...
     }
        private void abrirOpenGL(){
          ....
          imagenes.enableRenderer(IRenderer.RENDERER_OPENGL,IRenderer.MODE_OPENGL);
          imagenes.disableRenderer(IRenderer.RENDERER_SOFTWARE);
     }
     private void cerrarBuffer(){   ....    }
     ......
```

Para mostrar todos los objetos 3D (representar el escenario) del juego la aplicación tiene el siguiente método:

```
    private void dibujarPantalla(){
        try{
           imagenes.clear();
           synchronized(mundo){
              mundo.renderScene(imagenes);
           }
           mundo.draw(imagenes);
           imagenes.update();
           if (software)
              imagenes.display(gFrame,0,0);
           else
              imagenes.displayGLOnly();
        }
            catch(Exception e){}
     }
```

Por otro lado, será la clase que enlazará ‘MenuObjetos’ con ‘Juego’, para poder utilizar los eventos producidos por el teclado, y así por ejemplo poder cambiar la escala de un objeto 3D, moverlo antes de colocarlo fijamente sobre el escenario, colocarlo, saber cuál es el directorio desde el que se parte para poder buscar los archivos 3D, docentes, y mostrar todas las ventanas que genera la parte de edición del guión (clase ‘MenuObjetos’)...

```
     public void dibujarMenu(){
        imagenes.clear();
        menu.setMundo(mundo);
        menu.setTeclado(tecla);
        if(menu.getCargar()){
           if(menu.getPuerta())
              menu.cargarObjeto("puerta");
           else if(menu.getAscensor())
              menu.cargarObjeto("ascensor");
           else if(menu.getPersonaje())
              menu.cargarObjeto("personaje");
        }
        try{
           synchronized(mundo){
              mundo.renderScene(imagenes);
           }
           mundo.draw(imagenes);
           if(menu.isVisible()){
              mouse.show();
```

```
            menu.evaluateInput(mouse, keyMapper);
            menu.draw(imagenes);
         }
         else
            mouse.hide();
         imagenes.update();
         if (software)
            imagenes.display(gFrame,0,0);
         else
            try{
               imagenes.displayGLOnly();
            }
                catch(Exception e){}
      }
          catch(Exception e){
            e.printStackTrace();
         }
   }
   public void decrementar(){    ...      }
   ....
```

Dentro del método anterior (`dibujarMenu()`), cabe destacar la invocación "`menu.evaluateInput(mouse, keyMapper);`" donde evaluará los eventos producidos por el ratón y/o teclado, para poder saber si se ha pinchado sobre un botón que tiene programado un manejador de eventos para realizar una función o si se está escribiendo por teclado para mostrar el texto correspondiente.

Para poder ver la ventana de selección de sonido se utiliza el siguiente método:

```
   public void crearMenuSonido(){
        imagenes.clear();
        try{
           mundo.renderScene(imagenes);
          mundo.draw(imagenes);
           if(sonido.isVisible()){
              mouse.show();
              sonido.evaluateInput(mouse, keyMapper);
             sonido.draw(imagenes);
           }
           else
              mouse.hide();
           imagenes.update();
           if (software)
              imagenes.display(gFrame,0,0);
           else
              imagenes.displayGLOnly();
        }
            catch(Exception e){
              e.printStackTrace();
           }
   }
```

El método anterior será invocado siempre que se esté en el modo juego y la ventana correspondiente al sonido sea visible (se haya presionado la tecla 's' y no se haya dado respuesta, pinchando sobre el botón correspondiente).

Por lo tanto, en esta clase se reproducirá (en caso de estar activo) el sonido ambiente del juego.

Como ya se ha comentado, la clase implementa los métodos encargados de la representación por pantalla del juego, proceso que se lleva a cabo por medio de un hilo paralelo que ejecuta el método 'run' del objeto creado, que se encuentra basado en un bucle continuo en el que se ejecutarán rutinas de renderización y representación periódicamente, permaneciendo dormido el tiempo sobrante.

```
      public void run(){
         ....
         abrirOpenGL();
         while (vivo){
            tempor_actual=System.currentTimeMillis();
            if(guardar)
               edicion=menu.guardar();
            if(!edicion){
               if(sonido.isVisible()){
                  while(sonido.isVisible())
                     crearMenuSonido();
               }
               if (transic.RgetState())
                  dibujarPantalla();
            }
            else{
```

```
            if(menu.isVisible()){
               while(menu.isVisible())
                  dibujarMenu();
               imagenes.clear();
            }
            else
               dibujarPantalla();
         }
         try{
            tempor2=System.currentTimeMillis();
            if ((tempor2-tempor_actual)<MUESTREO)
               Thread.sleep(MUESTREO-tempor2+tempor_actual);
         }
             catch (Exception e){}
         if(dialogo.activo()){
            while(dialogo.activo()){
               dialogo.setSonido(sonido.getSonido());
               if(dialogo.isVisible()){
                  audio.close();
                  if(sonido.getSonido())
                     dialogo.sonido();
                  crearDialogo();
               }
            }
            if(!audio.getAbierto())
               if(sonido.getSonido()){
                 audio.open();
               }
         }
      }
      cerrarBuffer();
   }
}
```

- *Clase 'MenuObjetos'*

Esta es la clase con la que se realiza prácticamente toda la edición del guión XML del juego por lo que es nueva en la plataforma con respecto a la versión anterior.

Para poder llevar a cabo la edición de dicho archivo, en esta clase van apareciendo una serie de ventanas para ir obteniendo los datos que darán como resultado final el fichero. Por eso se puede separar cada ventana en un grupo: crear fase (en el que se pueden hacer otros subgrupos: colocar objetos, dar diálogo a los personajes), enlazar fases.

Para poder crear todas las ventanas se utilizan las clases que compone el apartado de "entorno de ventanas" y una vez creada la ventana, si es visible se muestra por pantalla gracias a la clase 'Representar'.

```
public class MenuObjetos extends Componente implements GUIListener {
     ...
     public MenuObjetos(Mundo mundo){   ...    }
     private void resetIndices(){    ...    }
     public void setMundo(Mundo mundo){   ...    }
     public boolean getFase(){
       return fase;
    }
    public void setTeclado(Teclado tecla){
       this.tecla=tecla;
    }
    public Teclado getTeclado(){
       return tecla;
    }
    public boolean esperar(){
        return espera;
    }
    public void setIdPersonaje(int id){
       this.id=id;
    }
    public boolean getCargar(){
       return cargar;
    }
    public boolean getPuerta(){
       return puerta;
    }
    public boolean getAscensor(){
       return ascensor;
    }
    public boolean getPersonaje(){
       return personaje;
    }
    private void limpiarVentana(){   ...    }
    private void dialogoCorto(String texto){  ...   }
```

```
    private void escribir(String texto){   ...   }
    private void informacion(String inf, boolean corto){  ...   }
    public void setVisible(boolean visible) {   ...   }
    public boolean isVisible() {   ...   }
    public void evaluateInput(MouseMapper mouse, KeyMapper keyMapper) {  ...   }
    public void draw(FrameBuffer buffer) {
       window.draw(buffer);
    }
    public void setFases(Vector fases){   ...   }
    private void numero(String msg){   ...   }
    private boolean comprobarDialogoPersonaje(int idPersonaje){   ...   }
    private boolean personajesDialogo(){   ...   }
    private Vector getIntervencionesOpcion(Vector inicio){   ...   }
    private void masFases(String texto){   ...   }
    private void masLlaves(){   ...   }
    private void otraOpcion(){   ...   }
    private String getPanel(int num, int orden){   ...   }
    private String getPanel(int numOpc){   ...   }
    private void elegirOpcion(int numOpc){   ...   }
    private void verFase(){   ...   }
    private Intervencion buscarInicio(Intervencion intermedia){   ...   }
    public boolean guardar(){   ...   }
    private void tipoVentana(char tipo){   ...   }
    private String getOpcion(Intervencion actual, int opc){   ...   }
    private Intervencion buscarIntervencionOpcion(Intervencion raiz, Intervencion opcion){  ...   }
    private Intervencion getSiguiente(Intervencion padre, int i){   ...   }
    private Intervencion getAnterior(Intervencion hijo){   ...   }
    private Intervencion buscarComun(Intervencion raiz, Intervencion com, Vector opciones, int pos){
       ...
    }
    private void nodosSinEnlazar(Intervencion raiz, Vector nodos, Vector opciones, int posicion){
       ...
    }
    private void recorrerOpciones(Vector opciones,Vector nodos, int posicion){   ...   }
    private int cuentaDialogos(Fase f){   ...   }
    private int getNumeroFasesEnlazadas(){   ...   }
    private Vector getIntervencionesFinalesIncompletas(Fase fase){   ...   }
    private Vector intervencionesIguales(Vector inter, Object comparada, int pos){   ...   }
    private Vector borraRepetidos(Vector inter){   ...   }
    public void guardar(String archivo,SimpleVector pos, int idFase){   ...   }
    public void setInicio(String archivo){   ...   }
    private String getNombre(){   ...   }
    public void setProbar(boolean probar){   ...   }
    public boolean getProbar(){   ...   }
    private int getNumDialogosTotales(){   ...   }
    private void finFase(){   ...   }
    private int comprobarDialogoPersonajes(){   ...   }
    public void setCamara(SimpleVector prota){   ...   }
    public String getFichero(){   ...   }
    public void setFichero(String fichero){   ...   }
    private void limpiarSeleccion(){   ...   }
    public boolean jugar(){   ...   }
    ...
}
```

Cuando se crea una fase, lo primero que pide la aplicación es que se introduzca un resumen identificativo (utilizado sólo a la hora de enlazar una fase con otra para que el desarrollador pueda identificar cada una rápidamente y si se ha creado la fase en el momento de arrancar la aplicación y enlazarlas). Y después de eso, muestra una ventana en la que se debe elegir el tipo de objeto 3D a colocar (puertas, ascensores y personajes) para pasar posteriormente a su selección.

```
    private void escribirFase(String txt){   ...   }
    private void pantallaObjeto(){   ...   }
    public void cargarObjeto(String objeto){   ...   }
    private void elegirObjeto(){   ...   }
```

El método '`cargarObjeto(String objeto)`' lo que hace es ir cambiando el archivo .3DS del objeto a seleccionar cada vez que se pulse sobre los botones "siguiente" o "anterior".

Una vez que se ha elegido el objeto a colocar sobre el escenario se puede mover o girar antes de colocarlo.

```
        public void aumentar(){     ...      }
        public void decrementar(){     ...      }
        public void avanzar(){     ...     }
        public void retroceder(){     ...      }
        public void subir(){     ...     }
        public void bajar(){     ...     }
        public void moverIzquierda(){     ...      }
        public void moverDerecha(){     ...      }
        public void girar(){     ...     }
```

```
        public void girarX(){     ...     }
        public void girarZ(){     ...      }
        public void colocar(){   ...   }
```

Después de colocar el objeto seleccionado, en el caso de ser un personaje, si tiene fotogramas que simulen que camina preguntará si ese personaje caminará en el juego (los fotogramas, como ya se ha comentado anteriormente, son diferentes archivos .3DS de ese personaje donde se diferencian por la posición de las piernas: un fotograma con una pierna un poco adelantada, otro con una pierna algo mas adelantada y la otra retrasada y viceversa), y, después de hacer que camine o no (o si no tiene esa posibilidad), preguntará si se quiere dar un diálogo (está la posibilidad de dar el diálogo justo cuando colocamos ese personaje o después de colocar otros personajes dentro de la misma fase siempre y cuando colisionemos con él), en el caso de ser un ascensor seguirá con una serie de ventanas necesarias para poder rellenar posteriormente el guión, o en caso contrario, preguntará si se quiere seguir colocando más objetos.

```
    private void movimiento(){   ...    }
    private void movimientoPersonaje(){   ...   }
    private void tipoAscensor(){   ...   }
    private void velocidad(){   ...   }
    private void masObjetos(){   ...   }
```

Para dar diálogo a los personajes, saldrá una ventana donde se debe elegir el tipo de intervención que tendrá en ese momento y cada vez que se incluya una intervención a la conversación preguntará si se quiere continuar con dicha conversación.

Si hay una intervención de tipo opción, si se quiere continuar la conversación primero preguntará si se quiere que tenga una parte en común. En caso afirmativo (de continuar con la conversación), irá preguntando si después de la última intervención introducida viene la parte común o no. Si viene la parte común y no hay ninguna de este tipo se introducirá normalmente (metiendo los datos), pero si se da el caso de que ya hay alguna intervención de ese tipo se preguntará si se ha introducido anteriormente o no. Si ya se ha puesto en la aplicación, en caso de que haya varias, mostrará una ventana de selección para que se elija la intervención que llegará en ese momento.

Cada vez que se cree una intervención nueva (que no sea de tipo test, final ni que identifique una parte común), en caso de haber archivos de audio, se tendrá la opción de dar sonido a dicha intervención.

Una vez dado el diálogo a un personaje, si hay otro sin diálogo asignado (porque se ha dicho que no ibamos a dar diálogo anteriormente, aunque posteriormente se pueda), antes de preguntar si se quieren colocar más objetos, preguntará si se ha terminado de dar diálogos.

```
    public void tipoDialogo(String txt){  ...   }
    private void texto(String txt){  ...   }
    private void tema(){  ...   }
    private int buscaTema(){  ...   }
    private int busca(Intervencion actual, int cantidad, int tipo){   ...   }
    private int cuentaTests(String directorio){  ...   }
    private int buscaTest(){  ...   }
    private String getArchivoTest(String directorio, int pos){  ...   }
    private void test(){  ...   }
    private void numPreguntas(){  ...   }
    private void numAprobar(String txt){  ...   }
    private void aprobar(){  ...   }
    private void suspender(){  ...   }
    private void textoConOpciones(String msg, String p, String op1, String o2, String o3, String o4,
String o5){    ...    }
    private void numeroOpciones(String msg){   ...   }
    private String getNoPadre(Vector inicio){   ...   }
    private void opcionConversacion(String opcion){   ...   }
    private void puntoComun(){   ...   }
    private void llegaComun(){   ...   }
    private void introducirPuntoComun(){   ...   }
    private void comprobarComun(){   ...   }
    private void buscarPuntoComun(){   ...   }
    private Intervencion buscarComun(Vector inicio){   ...   }
    private void sonido(){   ...   }
    private void masConversacion(){   ...   }
    private void masDialogo(){   ...   }
```

Si ya se tiene la conversación completa, se creará el diálogo, es decir, se enlazarán las intervenciones (se añaden los hijos).

```
    private void crearDialogos(Vector dia, Vector resultado){   ...   }
    private boolean intervencionOpciones(String op, Vector inicio){   ...   }
    private void buscarPadre(String padr, Intervencion itxt, int i, Vector inicio){   ...   }
```

Una vez creada una fase, está la posibilidad de guardar en un fichero lo que está editado.

```
    private void guardarFases(){    ...      }
    private void crearFichero(String nombre){   ...   }
```

Cuando no se quieran crear más fases, en caso de haber varias, hay enlazar unas con otras, para decir qué intervención final llevará a otra fase.

Por lo que para poder hacerlo, la aplicación va recorriendo por orden cada fase y así toma cada diálogo de una fase donde da a elegir uno (o una opción en caso de haber ramificación), y una vez elegido hay que decir a qué fase lleva, y así sucesivamente hasta recorrer todas las fases.

```
    private void elegirDialogoFase(){   ...   }
    private void faseOpciones(){  ...   }
    private void elegirFaseDialogo(){   ...   }
```

Para poder elegir correctamente tanto un diálogo como una fase, se puede visualizar el diálogo completo o el resumen fase.

```
    private String verResumenFase(){   ...   }
    private void verVentanaDialogo(){   ...   }
```

Una vez enlazadas todas las fases, la aplicación solicitará colocar al protagonista en el punto de partida del juego y para crear después el guión XML de juego.

```
    private void colocarProtagonista(){   ...   }
    private void crearFichero(){   ...   }
```

Por último destacar, que para poder manejar los eventos producidos en la ventana al presionar algún botón, hay implementar el método de la interfaz ‘GUIListener’:

```
    public void elementChanged(String label, String data) {  ...   }
```

Cabe decir que siempre que se pulse sobre un botón ‘Cancelar’, se mostrará la ventana correspondiente para asegurarse de que el desarrollador quiere cancelar realmente esa acción.

```
    private void principio(){    ...   }
    private void noTipoAscensor(){   ...   }
    private void noVelocidad(){  ...   }
    private void noMovimiento(){   ...   }
    private void noDialogo(){   ...   }
    private void noBuscarComun(){   ...   }
    private void noDialogoFase(){    ...   }
    private void noFaseDialogo(){  ...   }
    private void noOpcion(){   ...   }
    private void noFaseOpciones(){  ...   }
    private void informacionFaseDialogo(){    ...    }
   private void informacionDialogoFase(){   ...    }
```

- *Clase ‘Filtro’*

Con esta clase se pueden filtrar archivos que concuerdan con una determinada extensión. Utilizada por la clase ‘DirectoriosFiltrados’. Clase codificada por completo en la segunda versión.

```
    public class Filtro implements FilenameFilter{
      String extension;
       public Filtro(String extension){
         this.extension=extension;
      }
       public boolean accept(File dir, String nombre){
         extension=extension.toLowerCase();
         nombre=nombre.toLowerCase();
        return nombre.endsWith(extension);
      }
  }
```

- *Clase 'DirectoriosFiltrados'*

Gracias a esta clase se pueden obtener todos los archivos contenidos en un directorio determinado con la posibilidad de concordar con un filtro determinado. Para ello hay métodos con distinta funcionalidad.

Esta clase es muy utilizada por 'MenuObjetos' para poder obtener por ejemplo los archivos 3DS de las puertas.

Al igual que la clase anterior, ésta ha sido codificada por completo para llevar a cabo la evolución de la plataforma.

```
public class DirectoriosFiltrados{
   Vector v=new Vector();
   public Vector listarDirectorios(String directorio){    ...       }
   public Vector listarArchivos(String directorio){    ...    }
   public Vector listarDirectoriosInicio(String directorio, String filtro, String inicio){ ...  }
   public Vector listarDirectoriosFiltro(String directorio, String filtro){     ...      }
   public Vector listarDirectorios3DSFiltro(String directorio, String filtro){      ...    }
}
```

- *Clase 'Audio'*

Como la primera versión no soportaba sonido, esta clase ha sido codificada por completo en esta nueva versión. Es la clase con la que se reproduce el sonido de la aplicación. Dicha aplicación sólo soporta archivos cuyas extensiones sean: '.au', '.rmf', '.mid', '.wav', '.aif' y'.aiff'.

Si el archivo es válido se dispone a cargarlo para posteriormente reproducirlo.

```
public class Audio implements Runnable,  LineListener, MetaEventListener {
   ...
   public Audio(String fich) {
      if (fich != null) {
         if (fich.endsWith(".au") || fich.endsWith(".rmf") ||
            fich.endsWith(".mid") ||fich.endsWith(".wav") ||
            fich.endsWith(".aif") ||fich.endsWith(".aiff")){
            try {
               File file = new File(fich);
               if (file != null && file.exists())
                  fichero=file;
            }
                catch (SecurityException ex) {}
                catch (Exception ex) {}
         }
      }
   }
    public void open() {
      ...
      start();//arrancamos el hilo
   }
   public void close() {
      ...
      if (sequencer != null) {
        stop();//paramos el hilo
         sequencer.close();
      }
   }
    public boolean loadSound(Object object) {      ...      }
    public void playSound() {      ...     }
    ...
    public void run() {
      while (thread != null){
         if(loadSound(fichero)) {
            playSound();
         }
         ...
   }
}
```

Se genera en un hilo aparte donde reproduce el sonido continuamente en caso de ser ambiente o por el contrario sólo una vez, en caso de ser diálogo.

- *Clase 'MenuSonido'*

Esta clase es invocada (si se está jugando) cada vez que se pulse la tecla 's' (y no se muestre un diálogo de un personaje).

En ese caso se mostrará un diálogo que preguntará si se quiere sonido en el juego. En caso afirmativo se empezará a reproducir el sonido y en caso negativo el juego será "mudo".

```
public class MenuSonido implements GUIListener {
  private Texture backDrop = null;
  private Ventana window = null;
  private Boton aceptar = null;
  private Boton cancelar = null;
  private Etiqueta stateLabel = null;
  private Audio juke=null;
  private boolean sonido;
  public MenuSonido(Audio juke)throws Exception {
  //genera la interfaz gráfica de la ventana de diálogo
     ...
  }
   public void setVisible(boolean visible) {
     window.setVisible(visible);
  }
   public boolean isVisible() {
     return window.isVisible();
  }
   public void evaluateInput(MouseMapper mouse, KeyMapper keyMapper) {
     window.evaluateInput(mouse, keyMapper);
  }
   public void draw(FrameBuffer buffer) {
     window.draw(buffer);
  }
   public void elementChanged(String label, String data) {
     if (label.equals("Si")) {
        sonido=true;
        if(!juke.getAbierto())
           juke.open();
        setVisible(false);
     }
     if (label.equals("No")) {
        sonido=false;
        juke.close();
        setVisible(false);
     }
  }
  ...
}
```

También ha sido codificada por completo en esta versión.

- *Clase 'Teclado'*

Define los atributos y rutinas necesarias a la hora de llevar a cabo una gestión de los eventos producidos en la iteración con el teclado. De nuevo, toda aplicación requiere de un objeto perteneciente a esta clase que se encargue de este cometido.

Un hilo paralelo al principal será el encargado de la ejecución del método 'run' implementado en esta clase, método basado en un bucle continuo encargado de ejecutar rutinas periódicamente en las que se comprobarán posibles eventos de teclado, permaneciendo dormido el tiempo restante.

```
   private void mapearTeclado(){
     tecla=mapeador.poll();
     if(tecla!=KeyState.NONE){
        codigo_tecla=tecla.getKeyCode();
        if(tecla.getState()) pulsado=true;
        else pulsado=false;
        if (codigo_tecla==KeyEvent.VK_UP) adelante=pulsado;
        else if (codigo_tecla==KeyEvent.VK_DOWN) atras=pulsado;
        ....
     }
  }

  public void run (){
     ....
     while (vivo){
        tempor_actual=System.currentTimeMillis();
```

```
            if (transic.TPgetState()){
               mapearTeclado();
            }
            try{
               tempor2=System.currentTimeMillis();
               if ((tempor2-tempor_actual)<MUESTREO)
                  Thread.sleep(MUESTREO-tempor2+tempor_actual);
            }
                catch (Exception e){}
         }
         mapeador.destroy();
      }
```

El funcionamiento se encuentra basado en la activación de una variable determinada que refleja la eventualidad producida, a la que se accederá posteriormente desde el objeto 'Juego' o 'Protagonista' actuando consecuentemente a tales sucesos.

Esta clase ha sido modificada para llevar a cabo las funciones de edición, menú de sonido y guardar la partida.

4.1.1.2. Entorno de diálogo

Está constituido por todas las clases que implementan las herramientas para la sintetización de los cuadros de diálogos que construyen la interfaz de comunicación con los personajes. Dentro de este grupo se encuentran las clases 'Intervencion', 'Conversacion' y 'Dialogo' (figura 63).

- *Clase 'Intervencion'*

Se denomina 'intervención' a cada uno de los nodos que componen el árbol que caracteriza una conversación. De esta forma, un objeto de esta clase será tan solo una parte del diálogo que se mantendrá con un determinado personaje.

Esta clase contiene métodos con los que se pueden enlazar las intervenciones (añadir intervenciones hijas), obtener dichos hijos, saber si se está en una intervención final, así como la generación de las intervenciones test.

```
   public class Intervencion {
      ...
      public Intervencion (int nhijos,int tipo,int pregtest,Element raiz_test){   ...   }
      public Intervencion(int tipo){    ...     }
      public Intervencion(int tipo, int numOpc){    ...   }
      public String[] getOpciones(){    ...    }
      public Intervencion[] getRamificacion() {   ...   }
      public boolean addHijo(Intervencion hij){    ...     }
      public boolean esFinal(){ ...   }
      public void crearTest(){  ...   }
      ...
   }
```

Además contará con métodos para poder generar las distintas intervenciones cuando estemos editando el guión de juego y así poder generar correctamente el archivo XML correspondiente. Por lo que es una clase modificada con respecto a la primera versión.

```
      public void setPrincipal(String principal){   ...    }
      public String getPrincipal(){ ...  }
      public void setOpciones(Vector opciones){   ...      }
      public void setAnterior(Intervencion anterior){    ...     }
      public Intervencion getAnterior(){   ...   }
      public void setArchivo(String archivo){   ...    }
      public String getArchivo(){   ...   }
      public void setTema(int tema){   ...   }
      public int getIdTema(){   ...    }
      public void setNumeroPreguntas(int num){   ...    }
      public int getNumeroPreguntas(){   ...   }
      public void setSiguienteFase(int fase){   ...    }
      public int getSiguienteFase(){   ...   }
      public void setNota(int nota){   ...   }
      public int getNota(){   ...   }
      public void setIdPersonaje(int idPersonaje){   ...   }
      public int getIdPersonaje(){   ...    }
      public void setPadre(String padre){   ...   }
```

```
      public String getPadre(){   ...   }
      public void setId(int id){   ...   }
      public int getId(){   ...   }
      public void setIdComun(int idComun){   ...   }
      public int getIdComun(){   ...    }
      public void setSonido(String sonido){   ...   }
      public String getSonido(){   ...   }
```

- *Clase 'Conversacion'*

Todo diálogo tiene que estar caracterizado por un objeto del tipo 'Conversacion' encargado de llevar a cabo el control del progreso de la conversación. Esta clase implementa los atributos y métodos que permitan saber cual es la intervención inicial y actual de una conversación, obtener las intervenciones que derivan de una dada y todos aquellos procesos necesarios para gestionar la interacción con el personaje.

```
    public class Conversacion {
      public Intervencion actual;
      private Intervencion inicio;
      public Intervencion hijos[];
      public Intervencion comun[];
      ....
      public Conversacion (Intervencion ini,int num_test){
         ...
         colocarHijos();
      }
      private void colocarHijos(){     ....      }
      public Intervencion pasoAHijo(int i){   ....     }
      public void cerrarConversacion(){    ....      }
      public void setComun(Intervencion[] comun){    ...     }
      public Intervención[] getComun(){     ...      }
      ...
    }
```

Por lo tanto, es la clase encargada de enlazar las intervenciones y poder llevar a cabo el diálogo del personaje. Como ya se ha dicho, permite saber cuál es la intervención inicial de la conversación, así como cuáles son las intervenciones comunes que puede tener dicha conversación (en caso de tenerlas).

Gracias a esta clase se puede saber si un diálogo tiene una parte común o no, por lo que para llevar a cabo ésto, esta clase ha sido modifica.

- *Clase 'Dialogo'*

Clase que implementa la interfaz 'Runnable', esto indica que todo objeto creado a partir de la misma se ejecutará en un hilo paralelo al principal.

El ciclo de vida de un objeto diálogo comienza con la creación del mismo en la clase 'Juego' a partir del guión de la aplicación. Este proceso constructivo empieza con la creación de la conversación que caracteriza el diálogo en cuestión. Para ello se recorrerá de forma recursiva cada uno de los nodos que componen el árbol dando como resultado un nodo intervención, semilla de toda la conversación. Este nodo será asignado al anterior citado objeto conversación, cuyo constructor llevará a cabo las labores precisas para completar la configuración del mismo.

Una vez se tiene dicho objeto se pasará a implementar el objeto Diálogo, labor que desempeñará su constructor al que le pasaremos todos los parámetros necesarios.

Llegados a este punto se almacena el objeto Diálogo creado y se deja en espera hasta que sea reclamado por algún personaje en una fase determinada.

```
    public class Dialogo extends Frame implements Runnable{
       ....
      public Dialogo(int iddial, Conversacion conversacion){   ....      }
      public void setMaquinaEstados(TransicionEstados maquinaestados){   ....    }
      public boolean action (Event evento,Object arg){    ....      }
      private void construirFrame(Intervencion interv,int indice){   ...     }
      ...
  }
```

Producida la demanda se le asignará a dicho personaje el diálogo indicado, siendo responsabilidad de éste la creación del hilo correspondiente, que permanecerá inactivo mientras no se produzca ninguna colisión.

En caso de que tal colisión se produjese se activará dicha línea de ejecución, momento que coincidirá con el lanzamiento de la ventana de diálogo correspondiente. Seguidamente se volverá a desactivar una vez se cierre dicha ventana.

Para mostrar la ventana de diálogos la aplicación cuenta con los siguientes métodos (dependiendo del tipo de ventana construida):

```
    public void areaTexto(){    ...       }
    public void opciones(Intervencion interv, int num, boolean test, int indice){    ...     }
    public void cincoOpciones(Intervencion interv, boolean test, int indice){     ...    }
    public void cuatroOpciones(Intervencion interv, boolean test, int indice){     ...    }
    public void tresOpciones(Intervencion interv, boolean test, int indice){      ...     }
    public void dosOpciones(Intervencion interv, boolean test, int indice){     ...    }
    private void pantallaEscritura(String texto){      ...      }
}
```

Como en la versión anterior el diálogo se encontraba en una ventana emergente, gracias a la modificación de esta clase dicha ventana se muestra dentro de la aplicación. Además de poder soportar el tipo de pregunta de libre respuesta.

Por lo que además esta clase tiene métodos para poder representar las ventanas de diálogo dentro de la misma aplicación y poder escuchar los eventos producidos por el ratón y/o teclado.

```
    public void ocultar(){     ...      }
    public void setVisible(){     ...     }
    public boolean isVisible() {   ...     }
    public void draw(FrameBuffer buffer) {    ...     }
    public void evaluateInput(MouseMapper mouse, KeyMapper keyMapper) {   ...     }
```

Por otro lado, si está activo el sonido, cada vez que se cambie de intervención (con texto), se reproducirá el archivo de audio correspondiente a dicha intervención.

```
    public void sonido(){   ...    }
```

El hilo ejecutará de forma continuada un bucle de rutinas que se repiten de forma periódica, manteniéndose el tiempo restante dormido. Las rutinas implementadas estarán relacionadas con la representación de la animación en la ventana de diálogo que se ha lanzado.

```
    public void run() {
      ... //sonido del dialogo
      if (conversacion.actual.tipo==Intervencion.MODO_TEST)
         conversacion.actual.crearTest();
      construirFrame(conversacion.actual,ind_preg);
      while(vivo){
         tempor_actual=System.currentTimeMillis();
         try {
            tempor2=System.currentTimeMillis();
            Thread.sleep(MUESTREO-tempor2+tempor_actual);
         }
             catch(Exception e){}
      }
      activo=false;
      d.close();//apagamos el sonido del dialogo
   }
}
```

Finalmente es preciso hacer referencia a los eventos producidos por los botones de la ventana de diálogo. Ellos serán los que marquen el instante en el que se cambie de nodo dentro del árbol que caracteriza la conversación y ejecuten los métodos que adapten la pantalla al nuevo contenido.

```
    public void elementChanged(String label, String data){     ...      }
```

4.1.1.3. Entorno de Ventanas

Compuesto por todas las clases encargadas de llevar a cabo la representación de una ventana, así como la posibilidad de escuchar eventos producidos por el ratón y el teclado.

Como todas estas clases son necesarias para poder mostrar una ventana dentro de la aplicación, dichas clases han sido codificadas por completo en la evolución de la plataforma (figura 63).

- *Clase 'MousseMapper'*

Se trata de una pequeña clase que se utiliza para evaluar los eventos producidos por el ratón (si se ha presionado algún botón de éste).

Destacar de esta clase los métodos que obtienen las coordenadas donde se ha pinchado con el ratón para saber posteriormente si esas coordenadas están dentro de un objeto botón.

```
public class MouseMapper {
    public MouseMapper(int alto) {
      ...
       inicializacion();
    }
     public void hide() {     ...      }
     public void show() {      ...     }
     public boolean isVisible() {      ...      }
     public void destroy() {       ...      }
     public boolean buttonDown(int button) {      ...      }
     public int getMouseX() {       ...      }
     public int getMouseY() {       ...      }
     private void inicializacion() {      ...      }
}
```

- *Clase 'GUIComponent'*

Se trata de una clase abstracta puesto que consta de elementos sin implementar. Así el elemento en cuestión que herede de esta clase será el encargado de implementar dicho comportamiento particular.

Las clases que heredaran de esta abstracción son: 'Ventana', 'Boton', 'CampoTexto', 'Etiqueta' y 'MenuObjetos' (ésta última en caso de editar el guión).

Con esta clase se pueden posicionar en la aplicación las coordenadas de inicio de una ventana gráfica así como obtener dichas coordenadas (al igual que las coordenadas del objeto padre).

```
    public void setX(int x) {
         xpos = x;
    }
    public void setY(int y) {
         ypos = y;
    }
    public int getX() {
         return xpos;
    }
    public int getY() {
        return ypos;
    }
    public int getParentX() {
        if (superC!=null) {
          return superC.getX();
        }
    }
```

Esta clase se utiliza para aunar todos los elementos que contienen una ventana (para asociar todos los objetos que heredan de esta clase) o eliminar algún elemento.

```
      public void add(GUIComponent c) {
        comps.add(c);
         c.superC=this;
      }
       public void remove(GUIComponent c){
```

```
        if(comps.contains(c))
           comps.remove(c);
     }
```

Gracias a esta clase, se puede mostrar por pantalla la ventana completa que se ha generado.

```
    public void draw(FrameBuffer buffer) {
        if (visible)
           for (Iterator itty = comps.iterator(); itty.hasNext(); ) {
              GUIComponent c = (GUIComponent) itty.next();
             c.draw(buffer);
           }
        else
           buffer.clear();
     }
```

Está la posibilidad de determinar y saber si una ventana es visible o no, así en caso de ser visible puede ser mostrada por pantalla.

```
      public void setVisible(boolean visi) {
         visible=visi;
      }
       public boolean isVisible() {
         return visible;
      }
```

Además evalúa los eventos producidos por el teclado y ratón para saber si tiene que realizar una acción determinada.

```
     public boolean evaluateInput(MouseMapper mouse, KeyMapper keyMapper) {
        for (Iterator itty=comps.iterator(); itty.hasNext();) {
           GUIComponent c=(GUIComponent) itty.next();
           boolean has=c.evaluateInput(mouse, keyMapper);
           if (has) {
             return true;
           }
        }
        return false;
     }
```

- *Interfaz 'GUIListener'*

Interfaz que define el método que dirá lo que hay que hacer en caso de producirse un evento por el teclado o ratón.

```
    public interface GUIListener {
       void elementChanged(String label, String data);
   }
```

- *Clase TextBlitter*

Clase encargada de imprimir letras dentro de la aplicación (dentro del motor JPCT).
Para que se puedan mostrar las letras dentro de la aplicación, hay que indicar de dónde debe coger las letras necesarias.

```
private static Texture texture=new Texture("ventanas"+File.separatorChar+"font.gif");
```

Los métodos de esta clase son los siguientes:

```
public static void blitText(FrameBuffer buffer, String txt, int x, int y,char objeto){
     ...
     blitText(buffer, txt, x,y,maxX,maxY);
}
public static void blitText(FrameBuffer buffer, String txt, int x, int y, int maxX, int maxY) {
     ...
     buffer.blit(texture, iNum * ancho, yNum, x, y, ancho, alto, FrameBuffer.TRANSPARENT_BLITTING);
     ...
}
```

- *Clase 'Ventana'*

Esta clase es la encargada de mostrar la imagen que tendrá una ventana y así mostrarla por pantalla y evaluar los posibles eventos que se ha generado en ella o en algún componente que contiene.

```
public class Ventana extends GUIComponent {
    private Texture ventana=null;
     public Ventana(Texture ventana, int x, int y){     ...     }
     public void setTextura(Texture ventana){
       this.ventana=ventana;
    }
     public boolean evaluateInput(MouseMapper mouse, KeyMapper keyMapper) {     ...       }
     public void draw(FrameBuffer buffer) {      ...      }
}
```

- *Clase 'Etiqueta'*

Es la clase encargada de mostrar un texto fijo por pantalla. Además evalúa algún evento producido por algún componente (tipo botón por ejemplo) que compone la ventana.

```
public class Etiqueta extends GUIComponent {
    private String etiqueta = "";
    ...
    public Etiqueta(int xpos, int ypos) {       ...       }
    public void setY(int yp){
      this.yp=yp;
    }
     public void setX(int xp){
       this.xp=xp;
    }
     public void setTexto(String texto) {      ...       }
    ....
     public boolean evaluateInput(MouseMapper mouse, KeyMapper keyMapper) {
       return super.evaluateInput(mouse, keyMapper);
    }
     public void draw(FrameBuffer buffer) {       ...      }
}
```

Además contiene métodos utilizados para modificar el texto en caso de que dicha etiqueta esté dento de un "área de texto" y no se pueda ver todo el contenido, en cuyo caso si hay botones disponibles para acceder al texto no visible (subir o bajar), dicho texto será reemplazado por parte del texto no visible según el botón que se haya pulsado.

```
public void subir(){    ...   }
public void bajar(){    ...   }
```

- *Clase 'CampoTexto'*

Esta clase es la utilizada para poder introducir por teclado datos en la aplicación, mostrándolos por pantalla.

Un objeto de esta clase se crea pasando por parámetros las coordenadas (x, y) del campo así como su tamaño (ancho y largo).

Se puede escribir algo nuevo o borrar datos que se acaban de poner. Si se ha introducido mucho más texto del que puede mostrar la ventana se ocultará texto anterior y se podrá ver (como en la clase anterior) si están los botones adecuados.

Además, cada vez que se escriba algo y llegue al fin de línea, el cursor se cambiará a la línea siguiente y en caso de haber llegado al final del campo donde se ubicará será al principio de dicho campo.

```
public class CampoTexto extends GUIComponent {
  private String contenido = "";//contenido del Campo de Texto
  ...
   public CampoTexto(int xpos, int ypos, int xdim, int ydim) {       ...       }
   public String getTexto() {
     return total;
  }
   public void setY(int yp){
     this.yp=yp;
```

```
    }
     public void setX(int xp){
       this.xp=xp;
    }
     public void setTexto(String txt) {       ...      }
     public void borrar(){       ...       }
     public boolean evaluateInput(MouseMapper mouse, KeyMapper mapper) {      ...      }
     public void bajar(){       ...       }
     public void subir(){        ...       }
     public void draw(FrameBuffer buffer) {       ...           }
}
```

- *Clase 'Boton'*

Es la clase encargada de simular un botón dentro de la ventana para poder realizar una determinada acción tras pulsar en dicho botón.

Cada botón tiene un texto identificativo que puede ser mostrado o no por pantalla.

Puede ser un botón "simple" o un botón "selección", en el segundo caso si hay varios botones selección dentro de la misma ventana se podrá utilizar un atributo posición para indicar cuál de los botones selección ha sido pulsado y saber si está activo (en ese caso se mostrará "*" en el botón).

Para construir un botón, al igual que otros componentes, se deben indicar las coordenadas del plano dentro de la aplicación así como sus dimensiones (ancho y largo del botón).

```
public class Boton extends GUIComponent{
 ...
     public Boton(int xpos, int ypos, int xdim, int ydim) {       ...      }
     public void setX(int xpos){
       xp=xpos;
    }
     public void setY(int ypos){
       yp=ypos;
    }
     public void setTextoVisible(boolean hide) {
       textoVisible=hide;
   }
     public void setEtiqueta(String etiqueta) {
       this.etiqueta=etiqueta;
    }
     public String getEtiqueta(){
       return etiqueta;
    }
     public boolean getActivo(){      ...      }
     public void setPosicion(String posicion){
       this.posicion=posicion;
    }
     public String getPosicion(){
       return posicion;
    }
     ...
     public boolean evaluateInput(MouseMapper mouse, KeyMapper keyMapper) {      ...      }
     public void draw(FrameBuffer buffer) {     ...      }
}
```

Para que se puedan realizar diferentes acciones tras haber pulsado sobre un botón tenemos el siguiente método:

```
     public void setListener(GUIListener bl) {
       this.bl=bl;
    }
```

que a la hora de la creacción del botón debe ser invocado por él.

4.1.1.4. Entorno de Control

Compuesto por todas las clases encargadas de llevar a cabo el control del juego así como las labores de sincronización (figura 63).

➢ *Clase 'Fase'*

Todo objeto de esta clase representa un nodo caracterizado por la aparición de determinados personajes, puertas y ascensores, dentro del grafo que compone el juego.

Debido a esto la clase 'Fase' estará formada por un conjunto de arrays en los que se almacena la información referente a dicho estado de juego.

```
public boolean puertas[];
public boolean ascensores[];
public boolean personajes[];
public Dialogo dialog_person[];
public SimpleVector posicion_person[];
public SimpleVector posicion_puerta[];
public float rotacionY_person[];
public float rotacionY_puerta[];
```

En el caso de editar están los siguientes atributos:

```
private Personaje[] pers;
private Puerta[] puer;
private Ascensor[] ascen;
private String resumen;
```

La información almacenada en un objeto de esta clase será:

- Identificadores de personajes activos (en caso de jugar).
- Diálogos de cada uno de los personajes. En el caso de no tener ninguno se dejará como 'null' dicha posición.
- Posición de los personajes mencionados anteriormente.
- Identificadores de ascensores activos en este momento.
- Identificadores de puertas activas en este momento.
- Posición de las puertas mencionadas en el punto anterior.

En el caso de editar, la información será:

- Personajes de una fase.
- Puertas de una fase.
- Ascensores de una fase.
- Resumen que identifica a una determinada clase.

En el caso de jugar, con esta estructura se simplifica la labor del controlador del juego, encargándose este último únicamente de comprobar si se produjo un cambio de estado dentro de cierta variable interna que posee. En caso afirmativo acudirá al objeto 'Fase' cuyo identificador corresponda con el identificador destino especificado por la conversación, habilitando, deshabilitando y ubicando cada uno de los componentes que forman parte de la aventura.

En el caso de editar, tendremos todos los objetos correspondientes a una fase en un objeto 'Fase' para posteriormente poder generar el archivo XML correspondiente.

Adicionalmente a todos los atributos mencionados antes, están definidos los siguientes métodos.

```
public Fase (int id,int numpuertas,int numascen,int numperson){   ....   }
public Fase(int id, Puerta[] puertas, Ascensor[] ascensores, Personaje[] personajes, Dialogo[]
dialogo){    ...   }
public Dialogo[] getDialogos(){
    return dialog_person;
}
public Personaje[] getPersonajes(){
    return pers;
}
public void setPersonajes(Personaje[] pers){
     this.pers=pers;
}
public Puerta[] getPuertas(){
     return puer;
}
public void setPuertas(Puerta[] puer){
```

```
     this.puer=puer;
}
public Ascensor[] getAscensores(){
     return ascen;
}
public void setAscensores(Ascensor[] ascen){
     this.ascen=ascen;
}
public String getResumen(){
     return resumen;
}
public void setResumen(String resumen){
     this.resumen=resumen;
}
```

Por lo que como conclusión se puede decir que esta clase ha sido modificada para poder editar un guión de juego.

- *Clase 'TransicionEstados'*

Toda aplicación sintetizada desde esta plataforma consta de un objeto del tipo 'TransicionEstados', encargado del control de los cambios de fase que se llevan a cabo en juego.

Dicho control permite llevar a cabo los cambios de estados dentro de un juego de una forma correcta y ordenada, respetando siempre el sincronismo necesario para implementar una ejecución sin errores de la aplicación.

Esta clase define métodos y atributos que aportan viabilidad en tales operaciones. Algunos de los más destacables son:

- Atributo que identifica la fase actual.
- Atributo accesible por los objetos diálogos, configurado con el identificador de la siguiente fase a sintetizar
- Método encargado de realizar los cambios de fase. Este método hace uso de las dos variables anteriores de modo que, si aconteciese una diferencia entre las dos (no siendo la segunda de ellas -1, lo que supondría una vuelta a la misma fase), se procederá a configurar todas las variables necesarias por el controlador del juego para que, una vez recupere el control, lleve a cabo la deshabilitación y habilitación de componentes, el reposicionamiento de los mismos y la asignación de diálogos de la nueva fase sintetizada.

```
     private int estado_actual;
     public int estado_siguiente;
     private Fase fases[];
     private boolean abierto;
     private boolean TP;
     private boolean R;
     public Fase getEstadoSiguiente(){    ....      }
```

Además de estos tres componentes, en la clase se definen más herramientas relacionadas esta vez con el sincronismo en la ejecución de la aplicación.

Es obvio que este objeto consta de variables importantes, como aquellas que indican la siguiente fase a sintetizar. Éstas pueden ser accedidas desde diversos objetos implementados en diferentes hilos (en concreto por los objetos diálogo, y juego). Esta situación hace que se requiera implementar un cierto sincronismo que evite situaciones de carrera durante la ejecución de la aplicación. Para tal fin se crean métodos cerrojos dentro del objeto 'TransicionEstados' que bloquean al hilo que trata de acceder si ya hay uno accediendo a una de estas variables comprometidas.

```
      public synchronized boolean cerrarCandado(boolean espersonaje){
        if (espersonaje && ultimo==PERSONAJE){
           ultimo=PERSONAJE;
           return false;
        }
        else{
           if (abierto==false){
              try{
```

```
                wait();
            }
                catch(Exception e){
                  return false;
                }
        }
        abierto=false;
        if (espersonaje) ultimo=PERSONAJE;
        else ultimo=CONTROL;
        return true;
    }
}
public synchronized void abrirCandado(){
    abierto=true;
    notify();
}
```

Además de estos cerrojos se crean otros de menor importancia y sin ningún tipo de sincronización, encargados de bloquear los hilos que llevan a cabo la representación y el control de movimientos del protagonista. Estos métodos son empleados en momentos cruciales tales como un cambio de fase, en el que se bloquean todos los procesos citados anteriormente, o la conversación con un personaje, en la que únicamente se bloquearán los hilos encargados del movimiento del protagonista y de la gestión de eventos producidos por el teclado.

```
 public void TPsetState(boolean sst){
    TP=sst;
}
public boolean TPgetState(){
    return TP;
}
public void RsetState(boolean sst){
    R=sst;
}
public boolean RgetState(){
    return R;
}
public void setDD(boolean sst){
    despues_dialogo=sst;
}
public boolean getDD(){
    return despues_dialogo;
}
```

- *Clase 'Juego'*

El objeto de esta clase constituye el nodo principal de la aplicación, desempeñando labores de control del juego diseñado.

Según se esté en el modo edición o juego tendrá unas labores u otras.

La clase juego crea los métodos necesarios para llevar a cabo el análisis del guión del juego y la síntesis de cada uno de los componentes que aparecen en él.

Dependiendo de qué modo esté, tratará el documento XML de una u otra forma, puesto que si se edita puede que haya o no fases (y resto de elementos) y si se juega siempre habrá fases (y más elementos).

Si se está editando, se tratará el guión para crear los elementos que tiene cada fase de una forma que puedan ser utilizados los datos para generar posteriormente un nuevo guión de juego. En este caso habrá unas labores de control para comprobar el estado del teclado.

Si se está jugando, una vez se ha concluido con todo el documento XML se procede a inicializar todos los hilos creados, momento en el que se da vida a la aplicación y se pasa a realizar labores de control de la misma.

Estas labores de control van a ser las que se denominan con el apelativo de 'controlador del juego' y cuyos procesos son:

En caso de editar:

- Comprobar si se presionó alguna tecla y en ese caso realizar la función programada, ya sea mover los objetos 3D, colocarlos…

En caso de jugar:

- Comprobar si se produjo un cambio de fase. Para tal cometido se utilizará el objeto de la clase 'TransicionEstados'.

- En caso afirmativo bloquear teclado, representación y protagonista para proceder a habilitar, deshabilitar, reposicionar los componentes de la escena correspondientes a la fase que se desea sintetizar, asignando los diálogos adecuados a cada uno de los personajes.
- Volver a desbloquear todos los hilos haciendo que la aplicación siga su curso normal.
- En caso de no haber agotado el tiempo asignado a tales rutinas, dormir el hilo durante un periodo equivalente al tiempo restante.

```
      private void jugar(){
        ....
        while (vivo){
           ...
           if(edicion){
              ...
           }
           if(!edicion){
              ...
           }
           ...
           if (tecla.getSalir())
              cerrarPrograma();
           try{
              Thread.sleep(Juego.MUESTREO);
           }
              catch(Exception e){}
        }
```

Adicionalmente, en este bucle se lleva a cabo una comprobación de los eventos producidos por el teclado como se ha comentado anteriormente.

De producirse alguno de estos eventos se procederán a ejecutar rutinas específicas para cada uno de ellos. Si se pulsó la tecla 'ESC' se procede a realiza un cierre ordenado de la aplicación. Este proceso irá finalizando todos y cada uno de los hilos paralelos en ejecución y liberando los recursos utilizados por la plataforma. Una vez concluidos tales procesos se finalizará la ejecución del hilo principal cerrando por completo la aplicación en curso.

Por otro lado, si se pulsó otra tecla y es el modo edición se llevarán a cabo métodos relacionados con los objetos 3D.

Es preciso indicar la presencia del método 'main' dentro de esta clase. Esta función se encarga de comprobar la sintaxis de la entrada introducida al ejecutar la plataforma, lanzando un error en caso de no ser válida. Finalizada la comprobación se crea un objeto de la clase 'Juego' que desencadena la ejecución inminente de la plataforma.

```
      public static void main(String[] args){
        boolean edic=false;
        if(args.length==1){
           if(args[0].equals("-juego")){
              Juego jue=new Juego(false);
           }
           else if (args[0].equals("-edicion")){
              Juego jue=new Juego(true);
           }
           else{
              System.out.println("Error de parámetros");
              System.out.println("java [-juego] || [-edicion]");
              System.exit(0);
           }
        }
        else{
           int i=JOptionPane.showConfirmDialog(frame, "¿Quieres editar el juego?", "Edición",
JOptionPane.YES_OPTION);
           if(i==0)edic=true;
           Juego jue=new Juego(edic);
        }
     }
```

La sintaxis de la llamada a la plataforma responde al siguiente modelo:

```
java Juego [-juego || -edicion]
```

Para finalizar, se puede comprobar la presencia de un campos opcionales (-edicion o -juego). Si el parámetro es "-edicion" indica que se está en el modo edición; mientras por el contrario, si el parámetro es "-juego" indica que lo que se quiere hacer es jugar. Si no hay

ningún parámetro opcional, la aplicación preguntará si se quiere editar, en caso de decir que no, pasará al modo juego.

Una vez que arranque la aplicación, siempre pedirá que se abra el archivo XML guión del juego, ya sea para editar o para jugar.

4.1.1.4. Entorno Docente

Al igual que pasaba con el entorno de ventana, este entorno es una evolución de la plataforma, por lo que todas las clases que lo componen han sido codificadas por completo en esta versión (figura 62).

- *Clase 'Pregunta'*

Es la clase que contiene todos los elementos necesarios para crear una pregunta de un test.

```
private String enunciado;
private Vector opciones;
private Vector respuestas;
private String tipoPregunta;
```

Los métodos que tienen esta clase son los constructores, los que dan acceso a los atributos y con los que se pueden modificar, asi como los que permiten añadir una opción y respuesta a la pregunta.

```
public Pregunta(String enunciado, String tipo){          ...      }
public Pregunta(String enunciado){          ...      }
public void addOpcion(String opcion){
  opciones.addElement(opcion);
}
public void addRespuesta(String opcion){
  respuestas.addElement(opcion);
}
public void setEnunciado(String enunciado){
 this.enunciado=enunciado;
}
public void setOpciones(Vector opciones){
  this.opciones=opciones;
}
public void setRespuestas(Vector respuestas){
  this.respuestas=respuestas;
}
public Vector getRespuestas(){
  return respuestas;
}
public Vector getOpciones(){
  return opciones;
}
public String getEnunciado(){
  return enunciado;
}
public String getTipoPregunta(){
  return tipoPregunta;
}
public void setTipoRespuesta(String tipo){
  this.tipoPregunta=tipo;
}
```

- *Clase 'Enunciado'*

Es la clase con la que se añade/modifica una pregunta al test.

Dicha clase es una aplicación swing por la que nos muestra una ventana desde la que se introducen los datos necesarios para la nueva pregunta. Por lo tanto, como toda ventana que se crea, tendrá el método manejador de los eventos producidos al presionar un botón de ésta y en ese caso se llevará a cabo unas determinadas acciones en función del botón presionado.

Esta clase también será utilizada para visualizar la información que contiene la pregunta, y este caso los campos en los que se podía añadir/modificar información no serán editables.

```
public class Enunciado extends JDialog implements ActionListener{
   ...
   private int index;
   private boolean modif, editable;
   public Enunciado(NuevoTest n){       ...      }
   public Enunciado(NuevoTest n, Pregunta p, int index){      ...     }
   public Enunciado(NuevoTest n, Pregunta p, int index, boolean editable){      ...     }
   public void iniciar(Pregunta preg){      ...     }
   public void actionPerformed(ActionEvent e){     ...     }
}
```

- *Clase 'Test'*

Clase con la que se genera un objeto que puede almacenar varias preguntas y donde se indica cómo se llamará y dónde se va a generar el fichero XML correspondiente al test.

Con esta clase se pueden acceder a los atributos ya nombrados.

```
public class Test{
  String fichero;
  Vector preguntas;
  public Test(){
     preguntas=new Vector();
  }
   public Test(Vector preguntas){
     this.preguntas=preguntas;
  }
   public void setPreguntas(Vector preguntas){
     this.preguntas=preguntas;
  }
   public void setNombre(String nombre){
     fichero=nombre;
  }
   public String getNombre(){
     return fichero;
  }
   public Vector getPreguntas(){
     return preguntas;
  }
   public int getNumPregunta(){
     return preguntas.size();
  }
}
```

- *Clase 'NuevoTest'*

Clase con la que se crea/modifica/ve un test en la aplicación.

Esta clase muestra una ventana swing.

Todas las preguntas del test que se están viendo/creando serán visualizadas en la tabla que tendrá la ventana.

Desde esta clase, tras la selección de una pregunta de la tabla anteriormente mencionada, se puede, según el botón presionado, ver/modificar la pregunta anteriormente seleccionada. Además se puede añadir/borrar preguntas al test, así como dar nombre al fichero donde se almacenarán los datos XML del test.

```
public NuevoTest(Docencia f){   ...   }
public NuevoTest(Docencia f, Test tem, int index, boolean editable){  ...   }
public void iniciar(){  ...  }
public void addPregunta(Pregunta p){   ...   }
public void dibujarTabla(){   ...   }
public void borrar(int index){   ...   }
public void modificar(int index,Pregunta p){   ...   }
public Pregunta getPregunta(int index){   ...   }
private Test guardar(){   ...  }
public void actionPerformed(ActionEvent e){   ...   }
```

A la hora de guardar un archivo test, para que la aplicación pueda saber que ese archivo es un test sin necesidad de abrirlo, el nombre que tendrá dicho archivo será "test+nombre que se le haya dado por teclado", antes de añadir "test" se comprobará si el nombre que se ha introducido por teclado no empieza ya por test para evitar la repetición.

Por otro lado, siempre que se añada/borre una pregunta deberemos actualizar la tabla ("`dibujarTabla()`").

- *Clase 'ModeloTablaTest'*

Es la clase que define el modelo que tendrá la tabla de la clase anterior y mostrará los datos correspondientes. Así se podrá mostrar los datos referentes al enunciado de una pregunta.

Por otro lado, si se pincha sobre un elemento de la tabla se seleccionará la fila correspondiente.

```
public class ModeloTablaTest extends AbstractTableModel {
   private String columNames[];
   private Object datos[][];
   public ModeloTablaTest(Vector d){
      String atri[]={"Enunciado"};
      setColumNames(atri);
      int colNo=atri.length;
      Object [][] dato = new Object[d.size()][colNo];
      if(d.size()>0){
         for(int i=0;i<d.size();i++){
            Pregunta t=(Pregunta)d.elementAt(i);
            dato[i][0]=t.getEnunciado();
         }
      }
      datos=dato;
   }
   ...
}
```

Puesto que hereda de la clase 'AbstractTableModel', hay que implementar los siguientes métodos para llevar a cabo acciones tales como la selección de fila, etc.

```
    public Class getColumnClass(int columnIndex) {
      return String.class;
   }
    public void setColumNames(String []cabeceras){
      columNames=cabeceras;
   }
   public String getColumnName(int colum){
     return columNames[colum];
   }
    public int getColumnCount() {
      return ( columNames.length );
   }
    public int getRowCount() {
      return ( datos.length );
   }
    public Object getValueAt( int fila,int col ) {
      return ( datos[fila][col] );
   }
    public void setValueAt( Object valor,int fila,int col ) {
      datos[fila][col] = valor;
      fireTableDataChanged();
   }
    public boolean isCellEditable( int fila,int col ) {
      return( false );
   }
```

- *Clase 'Cuestionario'*

Clase que almacena todos los objetos tests creados desde la aplicación o que hay en el directorio correspondiente.

```
public Cuestionario(Vector tests){
   this.tests=tests;
}
public Cuestionario(String fichero){
   ...
   construirCuestionario(raiz);
}
public void construirCuestionario(Element raiz){       ...     }
public Vector getTests(){   ...   }
public void setTests(Vector tests){    ...   }
```

Con esta clase se generan todos los archivos XML correspondientes a cada test.

```
public DocType crearDocType(){    ...   }
```

```
    public void crearFichero(){    ...   }
```

- *Clase 'Tema'*

Es la clase con la que se almacenan los datos correspondientes a un tema (contenido y título).

```
public class Tema{
     private String titulo, contenido;
      public Tema(String contenido){
        this.contenido=contenido;
     }
      public Tema(String titulo,String contenido){
       this.titulo=titulo;
        this.contenido=contenido;
     }
      public void setTitulo(String titulo){
        this.titulo=titulo;
     }
      public void setContenido(String contenido){
        this.contenido=contenido;
     }
      public String getTitulo(){
        return titulo;
     }
      public String getContenido(){
        return contenido;
     }
  }
```

- *Clase 'NuevoTema'*

Clase que muestra un objeto ventana swing con el que se visualiza, añade/modifica el contenido de un tema.

La ventana tiene un objeto Texto (que extiende de TextArea) que permite introducir el contenido del texto (o visualizarlo) donde está la posibilidad de limpiar dicho campo (si se añade o modifica el contenido, pero no visualizando).

Como todo objeto swing que implementa un '`ActionListener`' esta clase tiene el método con el que se manejan los eventos producidos al presiona un botón de esta clase, para así poder realizar como es debido las acciones mencionadas anteriormente.

```
    public class NuevoTema extends JDialog implements ActionListener{
      ...
      public NuevoTema(Docencia f){
        Tema tem=new Tema("");
        new NuevoTema(f, tem, 0, true);
     }
      public NuevoTema(Docencia f, Tema tem, int index, boolean editable){
        super((JFrame)f, "Tema", true);
        docencia=f;
        this.ver=!editable;
        indice=index;
        iniciar(tem, editable);
     }
      public void iniciar(Tema tem, boolean editable){
        //creamos la interfaz gráfica
     }
      public boolean addTema(){      ...     }
      public boolean modificarTema(){       ...     }
      public void borrar(){
        texto.borrar();
     }
      public void actionPerformed(ActionEvent e){
       //manejador de eventos
         if (e.getSource() == aceptar) {         ...       }
        else if(e.getSource()==borrar){
           borrar();
        }
        else if(e.getSource()==cancelar){    ...    }
     }
  }
```

- *Clase 'Texto'*

Es una clase, como se ha comentado en el apartado anterior, que extiende de 'TextArea' para poder tener la opción de limpiar (borrar) dicho campo, obtener el contenido correctamente o incluso modificarlo.

```
    public class Texto extends TextArea{
       String texto;//es el texto del TextArea
       public Texto(boolean editable, String aux){
        super("",10,68,TextArea.SCROLLBARS_VERTICAL_ONLY);
        texto=paramString();
        setTexto(aux);
        setEditable(editable);
     }
       public int getLongitud(){
        obtenerTexto();
        return texto.length();
     }
      private void obtenerTexto(){
        texto=paramString();
        int inicio=texto.indexOf("text=")+5;
        String t=texto.substring(inicio);
        int fin=t.indexOf(",editable,selection=");
        texto=t.substring(0, fin);
     }
      public String getTexto(){
        obtenerTexto();
        return texto;
     }
      public void borrar(){
        replaceRange("",0,getLongitud());
     }
      public void setTexto(String t){
        replaceRange(t,0,getLongitud());
     }
  }
```

- *Clase 'Temario'*

Es una clase con la que se obtiene todo el contenido de temas que tiene la aplicación, ya sea que se haya introducido en ese momento o que estuviera editado en el fichero XML de temas que haya en el directorio correspondiente (en el que se ha arrancado la aplicación).

```
    public class Temario extends DocType{
      private Vector temas;
      private String fichero;
       public Temario(Vector temas){
         this.temas=temas;
         Temario();
      }
       public Temario(){
fichero="."+File.separatorChar+"juegos"+File.separatorChar+"Geografia"+File.separatorChar+"xml"+File.
separatorChar+"curso.xml";
      }
       public void setNombre(String nombre){
         fichero=nombre;
      }
       public String getFichero(){
         return fichero;
      }
       public void setTemas(Vector temas){
         this.temas=temas;
      }
       public Vector getTemas(){
         return temas;
      }
       ...
    }
```

En el caso de arrancar la aplicación y que de antemano hubiera un fichero XML con el contenido de temas en el directorio (pasado como argumento en la ejecución de la aplicación), como ya se ha dicho antes, esta clase obtiene los datos temas de ese archivo.

```
       public Temario(String fichero){
          ...
          construirTemario(raiz);
       }
       public void construirTemario(Element temario){
```

```
         ...
         if (temario!=null){
            listaaux1=temario.getChildren();
            longlist=listaaux1.size();
            for (int i=0;i<longlist;i++){
               eleaux=(Element)listaaux1.get(i);
               id=eleaux.getAttributeValue("numero_tema");
               titulo="TEMA "+id;
               if (!(id.compareTo("")==0||titulo.compareTo("")==0)){
                  String contenido=eleaux.getText();
                  Tema tem=new Tema(titulo,contenido);
                  temas.add(tem);//agregamos el tema
               }
            }
         }
      }
```

A través de esta clase, se puede generar el fichero XML correspondiente a los datos que almacena esta clase.

```
       public DocType crearDocType(){
         DocType doc=new DocType("Curso",
"../"+File.separatorChar+"../"+File.separatorChar+"../"+File.separatorChar+"xml"+File.separatorChar+"curso.dtd");
         return doc;
      }
       public void crearFichero(){
      //creamos los elementos necesarios para generar el documento xml
         ...
         try{
         //creamos el documento xml de salida
            Document doc=new Document(raiz, crearDocType());
            FileOutputStream out=new FileOutputStream(fichero);
            XMLOutputter serializer = new XMLOutputter();
            serializer.output(doc, out);
            out.flush();
            out.close();
         }
             catch(Exception e){ System.exit(0);
            }
      }
```

Cabe destacar, que el fichero generado se guardará con el nombre “curso+el nombre dado por el desarrollador” (en caso de que el nombre que haya dado el desarrollador no empiece por curso).

- *Clase ‘Docencia’*

Es una aplicación swing con la que se añade/modifica/borra datos docentes anteriormente mencionados (temas y tests).

Para ello consta de dos partes, una para los datos temas y la otra para los datos tests.

Estos datos son almacenados en dos tablas (una para cada parte) donde se puede seleccionar una fila para visualizar/modificar el contenido que corresponde a la fila selecciona.

```
   public class Docencia extends JFrame implements ActionListener{
      ...
      private Temario temario;
      private Cuestionario cuestiones;
      public Docencia(){
        ...
        temas=new Vector();
        tests=new Vector();
        iniciar();
     }
      private void iniciar(){      // crea la interfaz grafica. ...      }
      public boolean getModifica(){
        return modiTema;
     }
      public boolean getModificaTest(){
        return modiTest;
     }
      public void agregarTema(Tema nuevo){
        nuevo.setTitulo("TEMA "+temas.size());
        temas.add(nuevo);//agregamos el tema
        dibujarTablaTema();//actualizamos la tabla
     }
```

```
    public void agregarTest(Test nuevo){
      tests.add(nuevo);//agregamos el test
      dibujarTablaTest();//actualizamos la tabla
    }
    public void modificarTema(int index, Tema tem){
      temas.setElementAt(tem,index);//reemplazamos el contenido
      dibujarTablaTema();//se actualiza la tabla
    }
    public void modificarTest(int index, Test tem){
      tests.setElementAt(tem,index);//reemplazamos el test
      dibujarTablaTest();//actualizamos la tabla
    }
    public void borrarTema(int index){
      temas.removeElementAt(index);
      dibujarTablaTema();
    }
    public void borrarTest(int index){
     tests.removeElementAt(index);
     dibujarTablaTest();
    }
    public Tema getTema(int index){
      Tema t=(Tema)temas.elementAt(index);
      return t;
    }
    public Test getTest(int index){
      Test t=(Test)tests.elementAt(index);
      return t;
    }
    public void actionPerformed(ActionEvent e){
      ...
    }
    ...
}
```

Siempre que se añada/borre algún tema o test se actualizará la tabla correspondiente para poder ver la entrada de esos datos.

```
public void dibujarTablaTema(){
     modelo_tema=new ModeloTabla(temas, true);
     tabla_tema.setModel(modelo_tema);
  }
    public void dibujarTablaTest(){
      modelo_test=new ModeloTabla(tests, false);
      tabla_test.setModel(modelo_test);
    }
```

Para finalizar, a través de esta clase se puede dar nombre al archivo XML de temas que generará la aplicación.

```
    private String guardar(){
      elegir=new JFileChooser("."+File.separatorChar+"juegos");
      int accion=elegir.showSaveDialog(this);
      String archivo=null;
      if (accion == JFileChooser.APPROVE_OPTION) {
         guardado=true;
         File file = elegir.getSelectedFile();
         try{
            archivo=file.getCanonicalPath();
            if(!archivo.endsWith(".xml"))
               archivo+=".xml";
            temario.setNombre(archivo);

        }
             catch(Exception ex){}
      }
      else
         archivo=guardar();
      return archivo;
    }
```

La sintaxis de llamada a la aplicación docente es:

```
java Docencia
```

- *Clase 'ModeloTabla'*

Como en el caso de la clase 'ModeloTablaTest', esta clase define los campos que tendrán las tablas de la clase del apartado anterior ('Docencia').

Por eso, en la parte de temas se visualizarán los datos que identifican a los temas, cuya cabecera se ha nombrado como título, pero en cada campo se mostrará "TITULO + num", donde num es el número de tema (orden) que corresponde a esa fila.

Mientras que en la parte de tests hay dos columnas, la primera corresponde al nombre de archivo que tiene ese test y la segunda al número de preguntas que tiene el fichero XML.

```
    public class ModeloTabla extends AbstractTableModel {
      private String columNames[];
      private Object datos[][];
      public ModeloTabla(Vector d, boolean tema){
         String atri[];
         if(tema){
            String aux[]={"Titulo"};
            atri=aux;
         }
         else{
            String aux[]={"Nombre Fichero", "Número Preguntas"};
            atri=aux;
         }
         setColumNames(atri);
         int colNo=atri.length;
         Object [][] dato = new Object[d.size()][colNo];
         if(d.size()>0){
            for(int i=0;i<d.size();i++){
               if(tema){
                  Tema t=(Tema)d.elementAt(i);
                  dato[i][0]=t.getTitulo();
               }
               else{
                  Test test=(Test)d.elementAt(i);
                  dato[i][0]=test.getNombre();
                  dato[i][1]=test.getNumPregunta();
               }
            }
         }
         datos=dato;
      }
       public Class getColumnClass(int columnIndex) {
         return String.class;
      }
       public void setColumNames(String []cabeceras){
         columNames=cabeceras;
      }
      public String getColumnName(int colum){
         return columNames[colum];
      }
      public int getColumnCount() {
         return ( columNames.length );
      }
       public int getRowCount() {
         return ( datos.length );
      }
       public Object getValueAt( int fila,int col ) {
         return ( datos[fila][col] );
      }
       public void setValueAt( Object valor,int fila,int col ) {
         datos[fila][col] = valor;
         fireTableDataChanged();
      }
       public boolean isCellEditable( int fila,int col ) {
         return( false );
      }
   }
```

- *Clase 'TablaRender'*

Con esta clase se configurar los objetos que representan todas las tablas de la aplicación.

Así en caso de pinchar sobre un objeto de una tabla (seleccionar) dicha fila correspondiente al objeto seleccionado se pondrá amarilla, y en caso contrario (si no está seleccionada) estará blanca (caso por defecto).

```
public class TablaRender implements TableCellRenderer{
       public TablaRender() {}
       public Component getTableCellRendererComponent(JTable table, Object value, boolean isSelected,
        boolean hasFocus, int row, int column)  {
         JLabel etiqueta = new JLabel();
         etiqueta.setHorizontalAlignment(SwingConstants.CENTER);
         etiqueta.setOpaque(true);
         if (isSelected)
            etiqueta.setBackground (Color.YELLOW);
         else
            etiqueta.setBackground (Color.WHITE);
         if (value instanceof String)
            etiqueta.setText((String)value);
         else if(value instanceof Integer)
            etiqueta.setText(value.toString());
         return etiqueta;
      }
   }
```

4.1.1.5. Generación de Guiones

Puesto que la generación de guiones (ya sea para editar o para guardar una partida) es una evolución de la plataforma, las clases que componen este apartado han sido codificadas por completo para llevar a cabo esta versión (figura 63) .

- *Clase 'Fichero'*

Es la clase encargada de generar los elementos mínimos y necesarios para poder crear un guión XML de juego, es decir gracias a esta clase se generan los elementos 'Escenario', 'Temporizador', 'Pantalla' y 'Protagonista' (pudiendo dar una posición sobre el mundo a este último elemento).

```
    public class Fichero{
      private SimpleVector camara=new SimpleVector();
       public void setCamara(SimpleVector camara){
          this.camara=camara;
       }
        public Element getProtagonista(Element raiz){      ...      }
        public Element getTemporizador(Element raiz){      ...      }
        public Element getPantalla(Element raiz){        ...     }
        public Element getEscenario(Element raiz){        ...     }
   }
```

- *Clase 'Guardar'*

Clase que hereda de 'Fichero' para poder obtener los elementos que genera dicha clase. Es la clase utilizada para crear un guión de juego XML en el caso de que se quiera guardar una partida.

```
    public class Guardar extends Fichero{
      private SimpleVector posicion;
      private DocType doc;
      public Guardar(SimpleVector posicion){
         this.posicion=posicion;
         setCamara(posicion);
      }
       public DocType crearDocType(){   ...      }
       private Element getFases(Element raiz){      ...      }
       private Element getPersonajes(Element raiz){      ...      }
       private Element getAscensores(Element raiz){      ...      }
       private Element getPuertas(Element raiz){       ...      }
       private void getTexto(Element raiz, Element padre){       ...      }
       private void getFinal(Element raiz, Element padre){       ...      }
       private void getOpcion(Element raiz, Element padre){       ...      }
       private void getTema(Element raiz, Element padre){       ...      }
       private void getTest(Element raiz, Element padre){       ...      }
       private void getSuspenso(Element raiz, Element padre){       ...      }
       private void getAprobado(Element raiz, Element padre){       ...      }
       private void tipoDialogo(Element raiz, Element padre){       ...      }
       private void getComun(Element raiz, Element padre){       ...      }
       private void getParte(Element raiz, Element padre){       ...      }
       private void getPartes(Element raiz, Element padre){       ...      }
```

```
    private Element getDialogos(Element raiz){        ...      }
    public void guardarPartida(String inicio, String nombreFich, int id){
      ...
      try{
         Document doc=new Document(juego, crearDocType());
         FileOutputStream out=new FileOutputStream(fichero);
         XMLOutputter serializer = new XMLOutputter();
         serializer.output(doc, out);
         out.flush();
         out.close();
      }
          catch(Exception e){}
   }
}
```

En el caso de guardar una partida, se crea el fichero igual que el guión con el que se estaba jugando y lo único que se modifica de él es la posición del protagonista y la fase actual (en caso de ser distintas).

- *Clase 'GeneraDocumento'*

Al igual que la clase anterior, esta clase también hereda de 'Fichero'.

Es la clase utilizada en el modo edición para poder generar el guión de juego XML según lo editado.

```
public class GeneraDocumento extends Fichero{
   private Fase[] fases;
   private DocType doc;
    public GeneraDocumento(Fase[]  fases){
      this.fases=fases;
   }
    public GeneraDocumento(Vector fas){        ...       }
    public GeneraDocumento(){}
    public DocType crearDocTypeJuego(){        ...       }
    private Element generaPersonajes(){        ...       }
    private Element generaAscensores(){         ...       }
    private Element generaPuertas(){         ...      }
    private void tipoIntervencion(Intervencion aux, Element padre){       ...       }
    private void creaDialogoComun(Intervencion intervencion, Element padre){       ...      }
    private void creaDialogoAprobado(Intervencion intervencion, Element padre){       ...      }
    private void creaDialogoSuspenso(Intervencion intervencion, Element padre){       ...      }
    private void creaDialogoFinal(Intervencion intervencion, Element padre){        ...      }
    private void creaDialogoOpcion(Intervencion intervencion, Element padre){        ...      }
    private void creaDialogoTest(Intervencion intervencion, Element padre){        ...      }
    private void creaDialogoTema(Intervencion intervencion, Element padre){        ...      }
    private void creaDialogoConversacion(Intervencion intervencion, Element padre){      ...     }
    private Element creaDialogoParteComun(Intervencion [] comun){     ...      }
    private Element generaDialogos(){       ...      }
    private Element generaFases(){      ...      }
    public String generaFichero(String nombre){
      ...
      try{
         Document doc=new Document(j, crearDocTypeJuego());
         FileOutputStream out=new FileOutputStream(nombre);
         XMLOutputter serializer = new XMLOutputter();
        serializer.output(doc, out);
         out.flush();
         out.close();
      }
          catch(Exception e){
            System.out.println(e.getMessage());
            System.exit(0);
         }
      return nombre;
   }
    public int cuentaTemas(String fichero){      ...        }
}
```

Además esta clase también es utilizada en el modo edición para saber el número de temas totales del que dispone la aplicación y así no exponer más temas de los que hay.

4.2.- DTD empleada en el desarrollo de juegos

En este apartado se va a desglosar la DTD con la que se implementarán los juegos susceptibles de ser sintetizados por la plataforma. Este texto constituye un manual de gran importancia para la comprensión de su funcionamiento, además de ser una referencia para el desarrollador donde conocer todas las posibilidades que ofrece el sistema en cuanto a la definición de juegos se refiere.

En este punto se caracterizarán cada uno de los elementos y atributos definidos en las DTD que definen las sintaxis de los archivos compatibles con la plataforma. Debido a esto es importante tener en cuenta el contenido de dichos archivos, que se encuentran expuestos en el anexo I.

4.2.1. Sintaxis para la creación del guión

Estos elementos son los que el desarrollador deberá introducir a mano para crear un guión lo más vacío posible:

- *Elemento 'Temporizador'*: Elemento optativo en el diseño del guión del juego. Sus atributos dan valor a los temporizadores utilizados para el control de la periodicidad de la ejecución de tareas en todos los objetos implementados en hilos independientes. La no aparición del mismo o de algunos de sus atributos, o la no validez del contenido presente en alguno de ellos, hará que el programa se sintetice con los valores por defecto asignados en dichos campos.
    - ***Atributos 'juego', 'teclado', 'representar', 'protagonista', 'personaje', 'ascensor' y 'dialogo'***: Atributos correspondientes a los objetos de las clases 'Juego', 'Teclado', 'Representar', 'Protagonista', 'Personaje', 'Ascensor' y 'Dialogo' respectivamente. Asigna el valor del atributo estático definido en estas clases con el que se rigen sus temporizadores. En caso de detectarse un error en el contenido de algunos de estos se dejará el valor por defecto asignado al temporizador y se lanzará un error leve.

- *Elemento 'Pantalla'*: Elemento optativo al igual que los atributos que lo componen, encargados de asignar valor a los atributos relacionados con la definición de la interfaz gráfica del juego. En caso de no ser definido este elemento se pasará a sintetizar el juego con los parámetros definidos por defecto para cada uno de los atributos que lo componen.
    - ***Atributo 'titulo'***: Variable opcional que da valor al título de la ventana donde se ejecuta el juego. En caso de no estar disponible se optará por un valor por defecto.
    - ***Atributo 'anchopantalla'***: Variable opcional que indica el ancho de la pantalla en píxeles. Todo valor asignado a esta variable debe ser potencia de dos para ser válido. En caso de no introducir ningún valor hacerlo de forma errónea, el depurador asignará un valor por defecto a este parámetro.
    - ***Atributo 'altopantalla'***: Variable opcional similar a la anterior, pero en este caso se especifica en píxeles el alto característico de la ventana de juego.
    - ***Atributo 'redimension'***: Variable opcional que indica la posibilidad de redimensionar la ventana de la aplicación, siempre y cuando se lleve a cabo una renderización software. Por defecto la redimensión esta inhibida.
    - ***Atributo 'collide_offset'***: Variable opcional que representa la distancia a partir de la cual se considera que el protagonista podría colisionar con un objeto determinado. Es un parámetro de gran importancia en el funcionamiento de las rutinas que detectan colisiones.

Este valor deberá ser definido de forma coherente a la velocidad del protagonista y el escalado del mundo donde nos encontramos. Esto permitirá que los métodos encargados de la detección de colisiones únicamente se activen en los momentos imprescindibles, no añadiendo carga de procesado innecesaria al juego. El valor por defecto definido es de 40 unidades.

- ***Atributo 'tune_for_outdoor'***: Variable opcional utilizada para preestablecer una configuración óptima para escenarios al aire libre. Inicialmente esta opción se encuentra desactivada.
- ***Atributo 'maximo_polyx_visibles'***: Variable opcional que indica el número máximo de polígonos visibles por escena. Esta variable constituye un límite en la renderización de la imagen. Todos aquellos polígonos que pasen la cota impuesta por esta variable no serán representados en pantalla, mostrando en la consola un mensaje de advertencia ante dicho suceso. Según la complejidad de la escena se asignará un valor u otro a esta variable, que por defecto cuenta con un valor de 21000 polígonos.
- ***Atributo 'glmipmap'***: Variable opcional que indica si el motor gráfico genera Mipmaps para todas las texturas del juego. Esta variable únicamente tiene sentido en caso de realizar una renderización hardware. Por defecto se le asigna un valor negativo.
- ***Atributo 'renderizacion'***: Variable opcional que indica el tipo de renderización utilizada. Existe la posibilidad de elegir entre dos valores 'software' y 'hardware', correspondiendo a una renderización por software o hardware respectivamente. En caso de optar por una sintetización de imagen llevada a cabo por hardware se comprobará que el equipo cuente con una tarjeta gráfica compatible, en caso contrario se volverá a la renderización software configurada por defecto.
- ***Atributo 'gltrilinear'***: Variable optativa que indica la posibilidad de aplicar una filtración trilinear de las texturas cargadas por el motor gráfico. Por defecto esta posibilidad esta inhibida. De nuevo, esta variable únicamente tiene sentido en caso de estar utilizando una renderización hardware.
- ***Atributo 'glcolordepth'***: Variable optativa que indica la cantidad de bits utilizado para la definición de los colores en caso de estar utilizando una renderización hardware. Si por el contrario se utiliza renderización software, se emplearán 24 bits para su definición. El valor por defecto para esta variable es de 16.
- ***Atributo 'glfixedblitting'***: Variable optativa que evita los problemas propios de algunas tarjetas gráficas al utilizar 16 bits de color y una renderización hardware. Ante tal situación configuraremos esta variable con valor afirmativo de modo que se haga un tratamiento y solución de dicha contrariedad.

- *Elemento 'Escenario'*: Elemento obligatorio en el que se engloban todos los componentes que diseñan la escena y sus características.
  - ***Elemento 'Mundo'***: Elemento optativo que define el objeto mundo, base de todo el entorno que posteriormente se irá creando. Los atributos y elementos que lo implementan definirán las propiedades características de la escena que vamos a sintetizar. En caso de no contar con dicho elemento en la definición del guión, se creará un objeto 'Mundo' por defecto, sin ningún foco de luz en su interior.
    - **Atributo 'activada_caida'**: Variable optativa que define si se encuentra activada la caída de luz a causa de la distancia o no. Por defecto estará desactivado.

- **Atributo 'caida_de_luz'**: Variable optativa complementaria a la anterior que define la velocidad con la que cae la luz a partir de una distancia dada. Cuanto mayor sea esta variable, mayor será la velocidad con la que un foco de luz determinado pierde eficiencia pasada una frontera. Por defecto dicha caída tendrá un valor de 100.
- **Atributo 'cobertura_luz'**: Variable optativa relacionada con las dos anteriores. Indica la distancia a partir de la cual un foco de luz empieza a perder intensidad. Por defecto la frontera se encontrará a 300 unidades.
- **Elemento 'Color_Luz'**: Elemento optativo cuyos atributos obligatorios indican las componentes RGB de la luz ambiente que caracteriza la escena. En el caso de no ser definida esta variable en el guión del juego, se dotará al mundo con una luz ambiente cuyas coordenadas sean (15,15, 10).
  - *Atributo 'r_luz_ambiente'*. Variable obligatoria cuyo valor define la componente roja de la luz ambiente.
  - *Atributo 'g_luz_ambiente'*: Variable obligatoria cuyo valor define la componente verde de la luz ambiente.
  - *Atributo 'b_luz_ambiente'*: Variable obligatoria cuyo valor define la componente azul de la luz ambiente.
- **Elemento 'Foco_Luz'**: Elementos optativos que determinan la presencia de un foco de luz cuyas características vendrán determinadas por los atributos que lo componen. Pueden existir tantos focos de luz como el diseñador considere oportuno.
  - *Atributo 'componente_rojo'*: Variable obligatoria cuyo valor define la componente roja del foco de luz.
  - *Atributo 'componente_verde'*: Variable obligatoria cuyo valor define la componente verde del foco de luz.
  - *Atributo 'componente_azul'*: Variable obligatoria cuyo valor define la componente azul del foco de luz.
  - *Atributo 'posicion_x'*: Variable obligatoria cuyo valor define la posición en el eje x del foco de luz.
  - *Atributo 'posicion_y':* Variable obligatoria cuyo valor define la posición en el eje y del foco de luz.
  - *Atributo 'posicion_z':* Variable obligatoria cuyo valor define la posición en el eje z del foco de luz.

- ***Elemento 'Entorno'***: Elemento obligado dentro de la definición del mundo. En este se especifican los parámetros necesarios para llevar a cabo la carga del modelo que conforma el escenario.
  - **Atributo 'archivo_3ds'**: Variable obligatoria que indica el archivo 3DS que contiene el modelo del entorno que se quiere sintetizar (ver nota 1).
  - **Atributo 'directorio_texturas'**: Variable obligatoria que indica el directorio donde se encuentran ubicados los archivos de tipo JPEG que definen las texturas del modelo de la escena.
  - **Atributo 'escala'**: Variable obligatoria que indica el tamaño del modelo cargado. De esta forma, una vez carguemos el entorno se

Nota 1: Debido a la diversidad de caracteres de separación implementados por las diversas plataformas ('\' en Windows, '/'en Linux) se optó por utilizar el carácter '-' para su implementación en nuestro lenguaje. De este modo, según sea la plataforma sobre la que se trabaje, la aplicación convertirá este carácter en su dual correspondiente. Esto supone una limitación en los nombres de los directorios a los que se hace referencia, no pudiendo contener ninguno de ellos el carácter '-'.

procederá a realizar un escalado del mismo, redimensionándole para que todos los objetos contenidos en él consten de un tamaño coherente

- *Elemento 'Protagonista'*: Elemento obligatorio dentro de la definición del guión, encargado de concretar las características del protagonista.
  - ***Atributo 'posicion_x'***: Variable obligatoria que indica la ubicación en el eje 'x' inicial del protagonista.
  - ***Atributo 'posicion_y'***: Variable obligatoria que indica la ubicación en el eje 'y' inicial del protagonista.
  - ***Atributo 'posicion_z'***: Variable obligatoria que indica la ubicación en el eje 'z' inicial del protagonista.
  - ***Atributo 'rotacion_y'***: Variable opcional que indica la rotación inicial respecto al eje 'y' del protagonista. Dicho valor indicará hacia dónde mira este personaje. En caso de no ser definida, se dejará al personaje tal y como se definió en el modelo.
  - ***Atributo 'altura'***: Variable obligatoria relacionada con la altura del protagonista. Este atributo define el tamaño del eje mayor de la elipse utilizada para la detección de colisiones y que engloba al actor principal.
  - ***Atributo 'grosor'***: Variable obligatoria relacionada con la anchura del protagonista. Este atributo define el tamaño del semi-eje menor de la elipse utilizada para la detección de colisiones, que engloba al actor principal.
  - ***Atributo 'velocidad_movimiento'***: Variable optativa que define la velocidad con la que se desplaza el protagonista. Viene determinada por el número de unidades que se desplazará el actor principal en cada uno de sus movimientos hacia delante. Por defecto se asignará un valor de 3 pero es conveniente definir este valor de acuerdo con el escalado del entorno.
  - ***Atributo 'velocidad_giro'***: Variable optativa que define la cantidad de radianes girados en un solo movimiento. Cuanto mayor sea el valor de esta variable, mayor será la velocidad de giro del protagonista. Por defecto su valor es de 0.05 y, al contrario de la variable anterior, este valor es aceptable en casi todos los casos.
  - ***Atributo 'velocidad_caida'***: Variable optativa que define la velocidad con la que el personaje cae. Al igual que la velocidad de movimiento, esta variable está relacionada con el escalado de la escena e indicará el número de unidades de desplazamiento vertical que caracterizan cada una de las iteraciones que componen la caída del protagonista. Su valor por defecto es de 4.

Los siguientes elementos son generados desde la aplicación, por lo que el desarrollador no necesita introducirlos a mano en el guón:

- *Elemento 'Personajes'*: Elemento optativo que contiene el reparto de personajes que intervienen en la historia. En el caso de no existir éste, se sintetizará una aplicación sin ningún personaje, de lo contrario el contenido de esta parte del guión estará compuesto por uno o más elementos 'Personaje' que definen a cada uno de los actores del juego.
  - ***Elemento 'Personaje'***: Elemento que describe a un personaje concreto, asignándole su posición inicial y aspecto. Dentro del elemento 'Personajes' existirá un número ilimitado (aunque mayor que uno) de estas entidades, construyendo así el reparto de actores presentes en el juego.
    - **Atributo 'id'**: Atributo obligatorio que define el identificador del personaje, por medio del cual será referenciado. Su valor vendrá

definido por un número entero positivo que será validado por el compilador de la plataforma, devolviendo un error grave en el caso de contener un valor incorrecto.

- **Atributo 'archivo_3ds'**: Variable obligatoria que indica el archivo 3DS que contiene el modelo del personaje que se quiere sintetizar (ver nota 1).
- **Atributo 'directorio_texturas'**: Variable obligatoria que indica el directorio donde se encuentran ubicados los archivos de tipo JPEG que definen las texturas del modelo del personaje.
- **Atributo 'escala'**: Variable obligatoria que indica el tamaño del modelo cargado. De esta forma, una vez cargado el modelo del personaje, se procederá a realizar un escalado del mismo, redimensionándole a un tamaño coherente con la escena.
- **Atributo 'posicion_x'**: Variable obligatoria que indica la ubicación en el eje 'x' inicial del personaje.
- **Atributo 'posicion_y'**: Variable obligatoria que indica la ubicación en el eje 'y' inicial del personaje.
- **Atributo 'posicion_z'**: Variable obligatoria que indica la ubicación en el eje 'z' inicial del personaje.
- **Atributo 'rotacion_x'**: Variable opcional que indica la rotación inicial respecto al eje 'x' del modelo que caracteriza al personaje. En caso de no ser definida, se dejará al personaje tal y como se definió en el modelo.
- **Atributo 'rotacion_y'**: Variable opcional que indica la rotación inicial respecto al eje 'y' del protagonista. Dicho valor indicará hacia donde mira este personaje. En caso de no ser definida, se dejará al personaje tal y como se definió en el modelo.
- **Atributo 'rotacion_z'**: Variable opcional que indica la rotación inicial respecto al eje 'z' del modelo que caracteriza al personaje. En caso de no ser definida, se dejará al personaje tal y como se definió en el modelo.
- **Atributo 'movimiento'**: Variable optativa que indica si el personaje definido en este elemento se encuentra estático o por el contrario realiza algún movimiento. Por defecto, y en el caso de no existir esta variable o encontrarse malformada, se opta por la inmovilidad.
- **Atributo 'velocidad'**: Variable optativa relacionada con el atributo anterior. Su valor indica el número de radianes que caracterizará el giro del personaje en cada iteración. Por defecto esta variable tiene un valor de 0.05 radianes.
- **Atributo 'eje'**: Variable optativa que indica si el personaje caminará sobre el eje X o Z. En caso de no aparecer, indicará que el personaje no se mueve de su sitio.
- **Atributo 'posFinal'**: Variable optativa relacionada con el atributo anterior. Su valor indica la posición final del recorrido del personaje. Una vez que el personaje llegue a la posición final indicada realizará un giro de 180º sobre su eje Y y caminará hacia la posición inicial, y viceversa. Por lo tanto realizará un recorrido lineal paralelo al eje indicado en el atributo anterior.

- *Elemento 'Ascensores':* Elemento optativo que engloba todos los ascensores que intervienen en la historia. En el caso de no existir, se construirá una aplicación sin ningún dispositivo de este tipo, de lo contrario el contenido de esta parte del guión estará compuesto por uno o más elementos 'Ascensor'.
  - ***Elemento 'Ascensor':*** Elemento que describe un ascensor concreto, asignándole su posición inicial y aspecto. Dentro del elemento 'Ascensores' existirá un número ilimitado aunque mayor que uno de estas entidades.
    - **Atributo 'id'**: Atributo obligatorio que define el identificador del ascensor, por medio del cual será referenciado. Su valor vendrá definido por un número entero positivo que será validado por el compilador de la plataforma, devolviendo un error grave en el caso de contener un valor incorrecto.
    - **Atributo 'archivo_3ds'**: Variable obligatoria que indica el archivo 3DS que contiene el modelo del ascensor que se quiere sintetizar (ver nota 1).
    - **Atributo 'directorio_texturas'**: Variable obligatoria que indica el directorio donde se encuentran ubicados los archivos de tipo JPEG que definen las texturas del modelo del ascensor.
    - **Atributo 'escala'**: Variable obligatoria que indica el tamaño del modelo cargado. De esta forma, una vez cargado el modelo del ascensor, se procederá a realizar un escalado del mismo, redimensionándole a un tamaño coherente con la escena.
    - **Atributo 'modo_funcionamiento'**: Variable optativa que indica el modo de funcionamiento característico del ascensor. Por defecto, en caso de no estar definida o de contener un valor no valido, el modo elegido será el alternativo. Los posibles valores que podemos asignar a este atributo son:
      - '*modo_alternativo*': El ascensor se moverá únicamente si el protagonista está colisionándolo o si se encuentra ubicado en una posición que no sea ni la primera planta ni la segunda. En otro caso permanecerá inmóvil.
      - '*modo_movimiento*': El ascensor se encuentra en continuo movimiento, subiendo y bajando alternativamente.
      - '*modo_planta1*': El ascensor permanece inmóvil en la planta uno mientras que no se produzca una colisión, caso en el que se desplazará a la planta dos para posteriormente volver a recuperar su posición inicial.
      - '*modo_planta2*': Similar al modo anterior, pero en este caso la posición inicial será segunda planta.
    - **Atributo 'posicion_x'**: Variable obligatoria que indica la ubicación en el eje 'x' del ascensor.
    - **Atributo 'posicion_y'**: Variable obligatoria que indica la ubicación en el eje 'y' del ascensor.
    - **Atributo 'posicion_z'**: Variable obligatoria que indica la ubicación en el eje 'z' del ascensor.
    - **Atributo 'posicion_planta1'**: Variable obligatoria que define la posición de una de las plantas entre las que se moverá el ascensor. Concretamente se definirá la posición 'y' en la que se desea que pare el movimiento alternante de este componente.

- **Atributo ‘posicion_planta2’**: Variable obligatoria que define la posición de la planta restante entre las que se moverá el ascensor. Concretamente se definirá la posición ‘y’ en la que se desea que pare el movimiento alternante de este componente.

  El orden en el que se definan dichas posiciones es indistinto en el funcionamiento del ascensor.

- **Atributo ‘velocidad’**: Variable obligatoria que define la cantidad de unidades de desplazamiento vertical que el ascensor realizará en cada una de las iteraciones que componen su movimiento. De idéntica forma a las velocidades definidas en los apartados anteriores, el valor tiene que adecuarse a la escala del entorno en el que se encuentre, definiendo así un movimiento coherente con la escena.

- **Atributo ‘rotacion_x’**: Variable opcional que indica la rotación inicial respecto al eje ‘x’ del modelo que caracteriza al ascensor. En caso de no ser definida, se dejará el ascensor tal y como se definió en el modelo.

- **Atributo ‘rotacion_y’**: Variable opcional que indica la rotación inicial respecto al eje ‘y’ del modelo que caracteriza al ascensor. En caso de no ser definida, se dejará el ascensor tal y como se definió en el modelo.

- **Atributo ‘rotacion_z’**: Variable opcional que indica la rotación inicial respecto al eje ‘z’ del modelo que caracteriza al ascensor. En caso de no ser definida, se dejará el ascensor tal y como se definió en el modelo.

➢ *Elemento ‘Puertas’:* Elemento optativo que contiene todos los objetos inertes que intervienen en la historia. En el caso de no existir éste, se construirá una aplicación sin ningún componente de este tipo, de lo contrario el contenido de esta parte del guión estará compuesto por uno o más elementos ‘Puerta’.

- ***Elemento ‘Puerta’:*** Elemento que describe un objeto inerte concreto, asignándole su posición inicial y aspecto. Dentro del elemento ‘Puertas’ existirá un número ilimitado aunque mayor que uno de estas entidades

  - **Atributo ‘id’**: Atributo obligatorio que define el identificador del elemento inerte (puerta), por medio del cual será referenciado. Su valor vendrá definido por un número entero positivo que será validado por el compilador de la plataforma, devolviendo un error grave en el caso de contener un valor incorrecto.

  - **Atributo ‘archivo_3ds’**: Variable obligatoria que indica el archivo 3DS que contiene el modelo del elemento inerte (puerta) que se quiere sintetizar (ver nota 1).

  - **Atributo ‘directorio_texturas’**: Variable obligatoria que indica el directorio donde se encuentran ubicados los archivos de tipo JPEG que definen las texturas del modelo del objeto inerte (puerta).

  - **Atributo ‘escala’**: Variable obligatoria que indica el tamaño del modelo cargado. De esta forma, una vez cargado el modelo del objeto inerte (puerta), se procederá a realizar un escalado del mismo, redimensionándole a un tamaño coherente con la escena.

  - **Atributo ‘posicion_x’**: Variable obligatoria que indica la ubicación en el eje ‘x’ inicial del objeto inerte.

  - **Atributo ‘posicion_y’**: Variable obligatoria que indica la ubicación en el eje ‘y’ inicial del objeto inerte.

- **Atributo 'posicion_z'**: Variable obligatoria que indica la ubicación en el eje 'z' inicial del objeto inerte.
- **Atributo 'rotacion_x'**: Variable opcional que indica la rotación inicial respecto al eje 'x' del modelo que caracteriza al objeto inerte. En caso de no ser definida, se dejará dicho objeto tal y como se definió en el modelo.
- **Atributo 'rotacion_y'**: Variable opcional que indica la rotación inicial respecto al eje 'y' del modelo que caracteriza al objeto inerte. En caso de no ser definida, se dejará dicho objeto tal y como se definió en el modelo.
- **Atributo 'rotacion_z'**: Variable opcional que indica la rotación inicial respecto al eje 'z' del modelo que caracteriza al objeto inerte. En caso de no ser definida, se dejará dicho objeto tal y como se definió en el modelo.

➢ *Elemento 'Dialogos'*: Elemento optativo en cuyo interior existirán uno o más elementos 'Dialogo', correspondientes cada uno de ellos a una conversación propia de alguno de los personajes creados. En caso de no estar definido no existirá ninguna conversación en el juego.

- ***Elemento 'Dialogo'***: Uno o más elementos de este tipo constituyen el contenido del elemento padre definido en el punto anterior. En el se especifican las características de cada uno de los diálogos que aparecerán a lo largo de la aventura, así como de la ventana emergente que lo muestra por pantalla.

  Este elemento está caracterizado por solo un elemento de tipo intervención (intervencion_tema, intervencion_texto, intervencion_test, intervencion_opcion, intervencion_label o intervencion_final) que constituirá el nodo raíz del árbol que define la conversación. De esta forma, este nodo intervención contendrá uno o varios nodos hijos y así sucesivamente.
  - **Atributo 'id'**: Atributo obligado que constituye el identificador por medio del cual referenciaremos el objeto 'Dialogo' creado a partir de los datos XML. Su valor estará compuesto por un número entero positivo.
  - **Elemento 'Intervencion_Texto'**: Elemento que define una intervención tipo texto (ver apartado 3.2.1.3.) dentro de la conversación. En su interior existen elementos que definen su carácter además de otro elemento de tipo intervención correspondiente a su nodo hijo dentro del árbol conversacional.
    - *Atributo 'sonido'*: Atributo optativo que indica el path del archivo de audio de esta intervención (ver nota 1).
    - *Elemento 'Texto_Area'*: Elemento obligatorio que define el texto que constituye la intervención, y que será expuesto en pantalla.
  - **Elemento 'Intervencion_Opcion'**: Elemento que define una intervención tipo opción (ver apartado 3.2.1.3.) dentro de la conversación. En su interior existen elementos que definen su carácter además de tantos elementos 'Opcion' como posibles bifurcaciones implementadas
    - *Atributo 'sonido'*: Atributo optativo que indica el path del archivo de audio de esta intervención (ver nota 1).
    - *Elemento 'Texto_Principal'*: Elemento obligatorio que define el enunciado de la propuesta implementada.

- *Elemento 'Opcion'*: Cada intervención del tipo 'Intervencion_Opcion' cuenta con uno o más elementos de este tipo, que a su vez implementan cada una de las ramas que dan lugar a la bifurcación. Su contenido está caracterizado por un elemento del tipo intervención que indica el nodo hijo del árbol conversacional por el que se optará en el caso de elegir esta opción.
    - Elemento 'Texto_Opcion': Elemento obligatorio que define el texto de la propuesta que caracteriza esta opción.

- **Elemento 'Intervencion_Test'**: Elemento que define una intervención tipo test (ver apartado 3.2.1.3.) dentro de la conversación. En su interior existen elementos que definen su carácter además de dos elemento (Aprobado y Suspenso) correspondientes a la superación o fracaso de la prueba expuesta.
    - *Atributo 'numero_preguntas'*: Variable obligatoria que define el número de preguntas que componen la prueba. La plataforma tomará estos ejercicios del documento XML pasado como parámetro, escogiendo éstos de manera aleatoria.
    - *Atributo 'nota_suspenso'*: Variable obligatoria que define el número de fallos a partir del que se considerará suspensa la prueba, desplazando la conversación hacia la intervención contenida en el elemento 'Suspenso'. En caso de aprobar se procederá del mismo modo pero esta vez con el elemento 'Aprobado'
    - *Atributo 'archivo_test'*: Variable obligatoria que indica el path del documento XML que contendrá la batería de preguntas entre las que elegiremos los ejercicios planteados en la prueba (ver nota 1).
    - *Atributo 'sonido'*: Atributo optativo que indica el path del archivo de audio de esta intervención (ver nota 1). En caso de editarlo la aplicación no muestra una elección de archivo audio, ya que la aplicación muestra preguntas aleatorias, por lo que es difícil que corresponda un archivo diálogo de este tipo con la pregunta correspondiente, además de que podría darse el caso de pasar de pregunta y el sonido seguir con la de la anterior o viceversa, por lo que para esta opción se implementa en un sonido ambiente.
    - *Elemento 'Suspenso'*: Elemento obligatorio cuyo único contenido es un elemento de tipo intervención que sintetice el siguiente nodo del árbol conversacional al que se desplazará la conversación en caso de suspender la prueba.
    - *Elemento 'Aprobado'*: Elemento obligatorio cuyo único contenido será un elemento de tipo intervención que sintetice el siguiente nodo del árbol conversacional al que se desplazará la conversación en caso de aprobar la prueba.

- **Elemento 'Intervencion_Final'**: Elemento que define una intervención de tipo final (ver apartado 3.2.1.3.) encargada de implementar las hojas del árbol conversacional y cuyo contenido define el identificador de la fase siguiente a la que se trasladará la máquina de estados.
    - *Atributo 'siguiente_fase'*: Variable obligatoria que identifica la fase siguiente a la que se trasladará el controlador del juego. En

la fase de compilación se cotejará esta variable con los identificadores de las fases implementadas y en caso de no existir dicho nivel se lanzará un error grave. En caso de optar por no realizar ningún cambio de fase, el valor de esta variable será '-1'.

- **Elemento 'Intervencion_Tema'**: Elemento que define una intervención de tipo tema (ver apartado 3.2.1.3.) dentro de la conversación. En su interior aparecerán elementos que definan su carácter además de otro elemento del tipo intervención correspondiente a su nodo hijo dentro del árbol conversacional.
  - *Atributo 'sonido'*: Atributo optativo que indica el path del archivo de audio de esta intervención (ver nota 1).
  - *Atributo 'archivo_tema'*: Variable obligatoria que define la ruta del documento XML con contenidos didácticos, del que nos serviremos para implementar el contenido de esta parte de conversación (ver nota 1).
  - *Atributo 'tema'*: Variable obligatoria que define el apartado elegido dentro del documento XML anterior. El compilador, tras comprobar la existencia y validez del archivo anterior, pasará a cotejar cada uno de los identificadores correspondientes a los diferentes apartados con el valor pasado en este atributo, verificándose la existencia de la información reclamada.
- **Elemento 'Intervencion_Comun'***:* Elemento que define una intervención común, en la que se compartirá parte de la conversación. Esta intervención sólo aparecerá en caso de que anteriormente haya habido una intervención de tipo opción, en dicho caso dos o más opciones podrían llegar al mismo punto de la conversación por caminos distintos. En la fase de compilación se cotejará esta variable con los identificadores de las intervenciones ParteComun implementadas y en caso de no existir dicha intervención se lanzará un error grave.
  - *Atributo idComun:* Atributo obligatorio que enlazará la intervención con la parte común correspondiente.
- **Elemento 'PartesComunes'**: Variable optativa que contiene todos los elementos ParteComun que tendrá la conversación de un determinado personaje.
  - *Elemento ParteComun*: Elemento obligatorio que tendrá las intervenciones comunes; un elemento de tipo intervención que sintetice el siguiente nodo del árbol conversacional al que se desplazará la conversación. En su interior existe otro elemento de tipo intervención correspondiente a su nodo hijo dentro del árbol conversacional.
    - *Atributo idComun*: Atributo obligatorio que define el identificador por el que se referenciará el objeto de tipo 'Intervencion_Comun'

- *Elemento 'Fases'*: Elemento de carácter optativo en la definición de un guión de juego, en cuyo contenido se encontrarán tantos elementos 'Fase' como niveles compongan nuestra aplicación (existiendo siempre al menos uno).
  - ***Atributo 'fase_inicial'***: Variable obligatoria que define el identificador de la fase inicial del juego, y cuyo valor será cotejado con todos los identificadores de los elementos 'Fase' definidos.

- ***Elemento 'Fase'***: Todo elemento 'Fases' esta formado por al menos una de estas entidades. En ella se especificaran los contenidos que podremos encontrar en la escena siempre que el juego se encuentre en este estado.
    - **Atributo 'id_fase'**: Variable obligatoria que define el identificador por el que se referenciará el objeto de tipo 'Fase' creado a partir de este elemento.
    - **Elemento 'Personaje_Activo'**: Todo elemento de tipo 'Fase' puede definir cualquier número de estas entidades, siempre y cuando sea menor o igual al número de elementos 'Personaje' implementados. Cada una de ellas representa a un personaje activo en esta fase del juego.
        - *Atributo 'id'*: Variable obligatoria que apunta al identificador del personaje que se quiere activar. Labores de compilación se asegurarán de que verdaderamente existe dicho personaje, devolviendo un error grave en caso negativo.
        - *Atributo 'dialogo_id'*: Variable obligatoria que apunta al identificador del diálogo que se quiere asignar al personaje en esta fase del juego. Labores de compilación se asegurarán de que verdaderamente existe dicho diálogo, devolviendo un error grave en caso negativo.
        - *Atributo 'posicionX'*: Variable optativa que define la posición del personaje respecto al eje 'x' en esta fase del juego.
        - *Atributo 'posicionY'*: Variable optativa que define la posición del personaje respecto al eje 'y' en esta fase del juego.
        - *Atributo 'posicionZ'*: Variable optativa que define la posición del personaje respecto al eje 'z' en esta fase del juego.
        - *Atributo 'rotacionY'*: Variable optativa que define la rotación del personaje respecto al eje 'y' en esta fase del juego.
    - **Elemento 'Ascensor_Activo'**: Todo elemento de tipo 'Fase' puede definir cualquier número de estas entidades, siempre y cuando sea menor o igual que el número de elementos 'Ascensor' implementados. Cada una de ellas representa a un ascensor activo en esta fase del juego.
        - Atributo 'id': Variable obligatoria que apunta al identificador del ascensor que se quiere activar. Labores de compilación se asegurarán de que verdaderamente existe dicho ascensor, devolviendo un error grave en caso negativo.
    - **Elemento 'Puerta_Activa'**: Todo elemento de tipo 'Fase' puede definir cualquier número de estas entidades, siempre y cuando sea menor o igual que el número de elementos 'Puerta' implementados. Cada una de ellas representa a un ascensor activo en esta fase del juego.
        - *Atributo 'id':* Variable obligatoria que apunta al identificador del objeto que se quiere activar. Labores de compilación se asegurarán de que verdaderamente existe dicho objeto, devolviendo un error grave en caso negativo.
        - *Atributo 'posicionX'*: Variable optativa que define la posición del componente respecto al eje 'x' en esta fase del juego.
        - *Atributo 'posicionY'*: Variable optativa que define la posición del componente respecto al eje 'y' en esta fase del juego.
        - *Atributo 'posicionZ'*: Variable optativa que define la posición del componente respecto al eje 'z' en esta fase del juego.

- *Atributo 'rotacionY'*: Variable optativa que define la rotación del componente respecto al eje 'y' en esta fase del juego.

4.2.2. Sintaxis para los contenidos didácticos

- *Elemento 'Curso'*: Elemento raíz del documento XML empleado para la definición de contenidos didácticos. Su interior estará compuesto cualquier número de elementos de tipo 'Tema'.
  - ***Elemento 'Tema'***: Bajo este elemento quedará definido el texto plano que se define en un apartado concreto.
    - **Atributo 'numero_tema'**: Identificador obligatorio con el que se referenciará el apartado que define el elemento. Su valor ha de ser un entero positivo.

4.2.3. Sintaxis para la batería de tests

- *Elemento 'Preguntas'*: Elemento raíz del documento XML que define una batería de preguntas de un tipo en concreto.
  - ***Elemento 'Pregunta'***: El elemento raíz 'Preguntas' puede contener cualquier número de estas entidades, siendo las encargadas de definir cada uno de los ejercicios que componen la batería de test. Cada uno de estos elementos estará compuesto por un elemento 'Enunciado' y por 'num_preguntas' elementos 'Respuesta'.
    - **Elemento 'Enunciado'**: Elemento que contiene el enunciado de la pregunta de test.
    - **Elemento 'Respuesta'**: Elemento que contiene una de las respuesta a la pregunta que caracteriza este ejercicio.
      - *Atributo 'solucion'*: Variable obligatoria que define si la respuesta planteada es cierta o falsa.

Con esto queda totalmente definida la sintaxis utilizada a la hora de crear juegos con la plataforma. Para obtener un mayor detalle en lo que a este lenguaje se refiere resulta aconsejable acudir al anexo I.

4.3.- Sistema de archivos

El árbol de directorios que estructura la aplicación resulta interesante puesto que proporciona los conocimientos necesarios a la hora de implantar un juego en la plataforma. Una vez instalada la aplicación se creará el siguiente árbol de directorios.

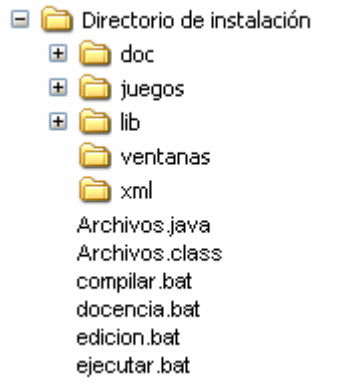

Figura 64: Árbol de directorios

Si se observa la figura se puede apreciar la aparición de los siguientes archivos y carpetas.

- *Carpeta 'doc'*: Contiene la documentación javadoc de la aplicación en formato HTML.

Figura 65: Árbol de directorios (II)

- *Carpeta 'lib'*: Contiene los archivos .jar que almacenan las librerías utilizadas por la aplicación entre las que se encuentran aquellas que implementan el motor jPCT, la API JDOM, y la API utilizada para llevar a cabo la renderización de las imágenes mediante OpenGL.

Figura 66: Árbol de directorios (III)

- *Carpeta 'xml'*: Contiene los archivos que implementan las DTD:
  - '***juego.dtd***': DTD del documento XML que define el guión del juego.
  - '***curso.dtd***': DTD de los documentos XML de contenidos didácticos.
  - '***test.dtd***': DTD de los documentos XML que conforman las baterías de pruebas.

- *Carpeta 'juegos'*: Carpeta en la que se ubican los ficheros que contengan cada uno de los juegos que se pueden ejecutar desde la plataforma. En el siguiente diagrama resulta posible observar más detalladamente la estructura de la misma.

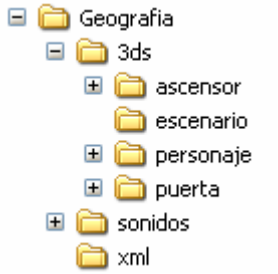

Figura 67: Árbol de directorios (IV)

Toda carpeta que engloba un juego concreto está ubicada sobre 'juegos'. En el caso del ejemplo será la carpeta 'Geografia' la que encierra la síntesis de dicha aplicación lúdica.

A partir de este ejemplo se analizará la estructura de directorios que se recomienda seguir en el despliegue de un juego. Así pues, todo directorio que almacena la implementación de un juego puede constar de los siguientes elementos:

- **Carpeta 'xml'**: Carpeta en la que se almacenarán todos los documentos XML que implementan un determinado juego. Entre éstos se encontrará un archivo que implemente el guión del juego, otro archivo que contenga un guión "vacío" (en el que sólo estén definidos los elementos necesarios para poder editar un juego desde cero) y el número necesario de archivos de contenidos y test.

  Dada la ubicación de estos documentos, la cabecera de todos ellos estará caracterizada por la siguiente línea:

  - Para el caso de juegos o guión de edición:

```
<!DOCTYPE Juego SYSTEM "..\..\..\xml\xxx.dtd">
```

  - Para el caso de tests:

```
<!DOCTYPE Preguntas SYSTEM "..\..\..\xml\xxx.dtd">
```

  - Para el caso de temas:

```
<!DOCTYPE Curso SYSTEM "..\..\..\xml\xxx.dtd">
```

  En ella se realiza un direccionamiento relativo a la DTD empleada en la validación del documento.

- **Carpeta '3ds'**: Carpeta en la que se ubican tanto los modelos como las texturas de los componentes gráficos que serán renderizados (ascensores, puertas, personajes y escenario). Dependiendo de su tipología, se almacenarán en una carpeta diferente como se puede ver en la figura 56.

  Los modelos reconocidos por la plataforma serán aquellos procedentes de la aplicación 3D Studio, caracterizados por la extensión 3DS.

  En cuanto a las texturas, la plataforma cargará todos los archivos con la extensión JPG ubicados en la dirección especificada en el guión.

- **Sonidos:** Carpeta que contiene los archivos de sonido del juego. El sonido ambiente está en está carpeta, mientras que los sonidos de los diálogos serán almacenados dentro de otra carpeta de este directorio llamada 'dialogos' (puede tener la lectura de cada intervención o un sonido por defecto en caso de no tener un archivo de sonido que corresponda con dicha lectura). Estos archivos pueden ser de distintas extensiones; la aplicación soporta las siguientes: .au, .rmf, .wav, .mid, .aiff, .aif.

- *Ventanas:* Carpeta que contiene las imágenes .gif que corresponden con las diversas ventanas que pueden aparecer en la aplicación.

- *Archivos .java*: Código fuente de la plataforma diseñada.

- *Archivos .class*: Archivos que contienen el bytecode que será interpretado por la máquina virtual de java a la hora de ejecutar la aplicación.

- *Archivo 'compilar.bat'*: Archivo ejecutable utilizado para la compilación del código fuente. Su contenido es el siguiente:

```
javac -classpath lib\jpct\jpct.jar;.;lib\lwjgl-1.0b3\lwjgl.jar;lib\jdom\jdom.jar;lib\jdom\xerces.jar *.java
```

- *Archivo 'docencia.bat'*: Archivo ejecutable con el que se pueden crear los documentos .xml correspondientes a la docencia (ya sean el archivo curso.xml que contiene los temas docentes, o los distintos test .xml).

```
java –Xmx800m -Djava.library.path=lib\lwjgl-1.0b3\ -classpath lib\lwjgl-1.0b3\lwjgl.jar;
lib\jpct\jpct.jar;.;lib\jdom\jdom.jar;lib\jdom\xerces.jar Docencia
```

- *Archivo 'edicion'*: Archivo ejecutable con el que podremos crear el guión del juego .xml.

```
java –Xmx800m -Djava.library.path=lib\lwjgl-1.0b3\ -classpath lib\lwjgl-1.0b3\lwjgl.jar;
lib\jpct\jpct.jar;.;lib\jdom\jdom.jar;lib\jdom\xerces.jar Juego -edicion
```

- *Archivo 'ejecutar.bat'*: Archivo ejecutable con el que se inicializa la plataforma para jugar.

```
java –Xmx800m -Djava.library.path=lib\lwjgl-1.0b3\ -classpath lib\lwjgl-1.0b3\lwjgl.jar;
lib\jpct\jpct.jar;.;lib\jdom\jdom.jar;lib\jdom\xerces.jar Juego -juego
```

5. – Validación de la aplicación

En este apartado se llevan a cabo una serie de pruebas que determinen el comportamiento de la plataforma. Todos estos ensayos convergen en la construcción de un juego que integra todas las posibilidades soportadas por la plataforma, pudiéndose comprobar su correcto funcionamiento en cualquier situación.

Esta fase de validación está dividida en las siguientes comprobaciones.

- *Prueba de usabilidad*: Comprobación de la efectividad de la plataforma a la hora de sintetizar un juego concreto. En esta fase se verificará una de las propiedades que se buscaban desde un principio, la sencillez y versatilidad a la hora de crear juegos.
- *Prueba de módulo*: Comprobación del funcionamiento de cada uno de los módulos que componen la aplicación. Esta fase se centra en evaluar el funcionamiento individual de cada una de las partes que constituyen la plataforma (compilador, controlador del juego, sintetizador del juego,…).
- *Prueba de integración*: Verificación del funcionamiento una vez se integran todos los módulos en los que se divide la plataforma.
- *Prueba de carga*: Prueba que evalua la robustez de la aplicación ante entradas sobredimensionadas.
- *Prueba de aceptación*: Fase en la que se verifica el funcionamiento de la aplicación de cara al usuario final además de comprobarse el grado de satisfacción de éste.

Se ha diseñado un juego para probar el perfecto funcionamiento de la plataforma, haciéndola pasar por todas las fases enunciadas anteriormente y comprobando los resultados obtenidos en cada una de ellas. En los siguientes apartados se detallará la estructura de dicho juego y su comportamiento frente a la fase de pruebas implementada.

5.1. - Demo para la evaluación final de la plataforma

En este apartado se hará referencia al juego de demostración implementado para comprobar el correcto funcionamiento de la plataforma, reflejando los resultados a los que se puede llegar con ésta.

Se arrancó la aplicación desde el ejecutable ‘docencia.bat’ para crear los archivos docentes que usará el juego.

Una vez creados los archivos docentes se editó el guión del juego partiendo de un archivo xml con los elementos mínimos para llevar a cabo la edición.

Se fueron generando las fases y posteriormente enlazándolas para llegar al guión final del juego.

El juego hace patente la vertiente educativa que posee la plataforma, sintetizando una aventura en la que el protagonista irá adquiriendo conocimientos de una pequeña introducción a la geografía española a medida que avance en ella.

Todo empieza ubicando al protagonista en una especie de laberinto en el que irá encontrando nuevos personajes a medida que avanza la historia. En un principio, se encontrará encerrado en una habitación en la que únicamente hay un personaje con el que poder hablar. Si opta por hablar con él, presentará la aventura y según avance ésta, se dejarán abiertan puertas que permitan inspeccionar cada uno de los rincones de la casa.

En el transcurso del juego, el protagonista se irá encontrando con personajes que le ofrecerán bifurcaciones en el devenir de la historia, teniéndose que optar por alguno de los caminos planteados. Esto posibilitará la inclusión de las prestaciones docentes que aportan un tutor inteligente en el transcurso del juego orientando al usuario por caminos que se ajustan a su perfil.

El tutorial virtual implementado con este juego consta de la siguiente estructura:

- *Personajes.*

  - ***Juan***: Compañero de travesía en el transcurso de la aventura. Será el encargado de asesorar al protagonista (correcta o incorrectamente) en ciertas fases del juego.

Figura 68: Juan

  - ***Libro de texto***: Tutor encargado de proveer información acerca de un tema del curso.

Figura 69: Libro de texto

  - ***Examinador***: Personaje encargado de evaluar los conocimientos del protagonista en ciertas fases de la historia, ofreciéndole consejos coherentes con el nivel que demuestre.

Figura 70: Examinador

  - ***Profesor***: Personaje que enseña al usuario un tema.

Figura 71: Profesor

- ***Hormiga***: Personaje que inicia en la aventura y guía.

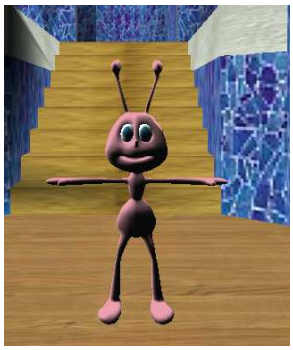

Figura 72: Hormiga

- ***Diploma***: Personaje que da por finalizada la aventura.

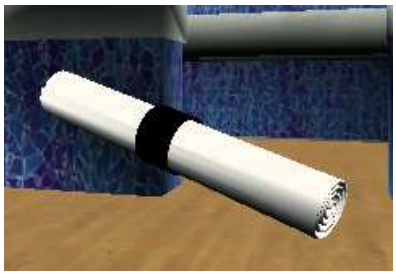

Figura 73: Diploma

- *Grafo del juego*: El grafo que caracteriza el juego viene reflejado en la siguiente figura. Los arcos reflejan las posibles transiciones de estado que se pueden realizar desde cada una de las fases.

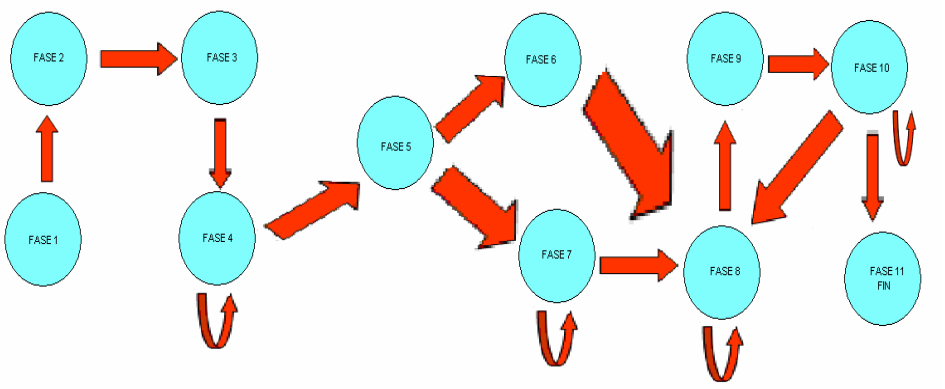

Figura 74: Grafo del juego

- *Fases*: En este apartado se concreta el contenido principal de cada una de las fases. Adicionalmente a lo mencionado en cada fase, existirán otros personajes encargados de entretener y asesorar al protagonista.
    - ***Fase 1***: Fase inicial del juego donde un personaje da la bienvenida al protagonista y le inicia en la aventura.
    - ***Fase 2***: En la cual habrá una bifurcación que llegará al mismo punto y llevará a la fase 3.
    - ***Fase 3***: Fase en la que está Juan y dice que está muy bien el tema que ya ha leído y deja leer al usuario.
    - ***Fase 4***: Fase de carácter docente donde se muestra el primer tema que imparte la aplicación. Puede llevar a la fase 5 o quedarse en la actual si no se comprende el contenido del tema.
    - ***Fase 5***: Accedida después de leer y creer comprender el documento que introduce al protagonista en la 'geografía española'. En ella el usuario se vuelve a encontrar con Juan que pregunta si le acompaña a la cafetería o si hace el examen.
    - ***Fase 6***: La cafetería, que es una fase de carácter lúdico para entretener al protagonista.
    - ***Fase 7***: Fase en la que se hace un examen que pondrá a prueba los conocimientos adquiridos en la fase 4. En caso de aprobar se pasará a la fase 8, mientras que en caso de suspender se tiene la posibilidad de volver a revisar el tema que se mostró en la fase 4 o quedarse en la fase actual para hacer el examen de nuevo.
    - ***Fase 8***: Fase en la que vuelve a enseñar un tema docente. A través de esta fase se puede pasar a la fase 9 si se cree comprender el tema o quedarse en la actual para repasar el tema de nuevo.
    - ***Fase 9***: Fase lúdica para entretener al usuario. Da acceso a la fase 10.
    - ***Fase 10***: Fase en la que se hace el último examen de la aventura. En caso de aprobar pasará a la fase 11, mientras que si se suspende se podrá hacer de nuevo el examen o revisar el contenido del tema volviendo a la fase 8.
    - ***Fase 11***: Fase final en la que se tiene acceso al diploma y quedan abiertas todas las habitaciones del escenario.

5.2. - Prueba de usabilidad

Esta prueba verifica la facilidad y versatilidad que proporciona la herramienta creada a la hora de sintetizar juegos. Se dispone de un guión simple para poder editar juegos y de otro guión complicado, caracterizado por múltiples encrucijadas que llevasen al protagonista a diferentes niveles donde poder ir desarrollando sus conocimientos acerca de un tema (en el ejemplo Geografia). Tal complejidad permite evaluar la usabilidad de la plataforma de cara a construir juegos que cumplen con los requisitos buscados desde un principio, simplicidad y potencia.

En este apartado hay que destacar, que a la hora de editar la aplicación sufre retardos (en el ordenador de pruebas) tras haber mostrado varias ventanas en la aplicación, para evitar este retardo se miró la posibilidad de algún colector de basura, limpiar el buffer donde se almacenan dichas ventanas (imágenes) pero aún así el retardo seguía apareciendo. Una de las formas en las que se evitaba dicho retardo (aunque latoso) sería según creamos una fase, guardarla y cerrar la aplicación para posteriormente arrancarla con el último guión generado.

Pero a pesar de eso, observando el producto final conseguido tras la ejecución de la aplicación diseñada, se puede concretar que la prueba de usabilidad se ha pasado con éxito, comprobando la efectividad de la plataforma construida.

5.3. - Prueba de módulo e integración

El guión desarrollado contiene todos los casos que se pueden dar en la síntesis de un juego, no dejando fuera ningún elemento posible. Esto permite la comprobación del comportamiento de cada uno de los componentes que intervienen en la plataforma a la hora de construir un juego.

Observando los resultados, y tras una exhaustiva comprobación del funcionamiento de cada uno de los componentes se pudo dar por válida las pruebas hechas a los diferentes módulos que componen la plataforma.

Paralelamente, se comprobó la corrección del compilador que precede a la síntesis del juego y por el que pasa todo documento XML. Para ello sirvió el guión diseñado anteriormente, el cual se fue deformando introduciéndole errores aleatorios que fueron detectados en todos los casos.

Por último se llevaron a cabo numerosas horas de iteración con el juego, con las que se pretendía probar el producto en conjunto. Tras esta sesión no se detectaron errores lo que daba por válida la prueba de integración.

5.4. - Pruebas de carga

Para llevar a cabo este cometido se implementó un guión, paralelo al definido antes, en el que apareciesen muchos modelos muy complicados, introduciendo una sobrecarga de información que dificultaba la renderización llevada a cabo por la aplicación.

Ante esta situación, la aplicación sufrió una merma en su velocidad debida a la falta de memoria RAM para almacenar todos los componentes propios de la escena. Esto se pudo solventar introduciendo una cota mayor de memoria RAM utilizada por la máquina virtual de java. Esto hace ver que, ante situaciones de sobrecarga, el contar con una definición correcta en la temporización usada por los hilos y suficiente memoria RAM, atenuará los efectos perjudiciales producidos por esta causa.

5.5. - Prueba de aceptación

Esta fase trata de evaluar la aceptación que existe por parte del usuario final en cuanto a las características de la plataforma se refiere.

Para tal fin se dispuso de un grupo de prueba encargado de evaluar la aplicación diseñada. Observando la valoración subjetiva de todos los componentes del grupo citado, que indicaban que la plataforma puede llegar a sintetizar juegos entretenidos y divertidos a la par que útiles en lo que a labores didácticas se refiere, se dio por validada esta última prueba.

6. – Conclusiones y trabajos futuros

6.1. – Conclusiones

El producto final que se ha obtenido da lugar a una herramienta capaz de crear juegos potentes de forma sencilla. La potencia de la plataforma ha sido comprobada con los resultados finales obtenidos tras la fase de pruebas a la que se sometió dicho juego en ese mismo capítulo.

Se ha obtenido una aplicación capaz de generar los ficheros docentes en formato XML siguiendo unas DTDs oportunas simplemente con la introducción de los datos necesarios.

Así mismo, esta aplicación es capaz de crear guiones de juegos según se vayan introduciendo objetos .3ds (obteniendo su posición y ángulos de giro) y dar diálogos a los personajes que se quiera, para posteriormente poder usar dicho guión en la aplicación de juego.

Para dar mayor atractivo a la aplicación, se ha introducido sonido, tanto ambiente como en los diálogos de los personajes, se maneja la posibilidad de dar movimiento (recorrido) a los personajes.

Para evitar la aparición de ventanas emergentes de los diálogos de los personajes, se consigue incrustar éstas dentro de la aplicación.

Además cabe la posibilidad de compartir intervenciones dentro de un diálogo.

Se ha podido comprobar que la plataforma de juego diseñada cuenta con los recursos para implementar aplicaciones lúdicas con connotaciones didácticas, capaces de andar los mismos pasos de aquellas iniciativas orientadas a implementar lo que se conoce como 'aprendizaje basado en juegos'.

El tutorial diseñado constituye un ejemplo de estas aplicaciones. Esta aplicación de juego sumerge al protagonista en un entorno virtual en el que se encontrará con varios personajes que le instruirán en un tema concreto (XML). De esta forma, el contenido didáctico que caracteriza el aspecto docente de la aplicación, queda difuminado a lo largo de una aventura entretenida que atraerá al usuario por si sola.

Por último reseñar que la aplicación se ha implementado con matices de ámbito docente, de modo que ofrezca todas las herramientas necesarias para implementar juegos de este tipo. Pero, a pesar de ello, este sistema es capaz de desarrollar cualquier tipo de juegos, siempre y cuando se ciñan a las herramientas proporcionadas.

6.2. - Trabajos futuros

Las líneas de trabajo futuro que surgen tras la implementación de esta plataforma tratan sobre el perfeccionamiento e incremento de prestaciones de la misma. Se pueden distinguir varias vertientes.

La primera de ellas se encuentra relacionada con la apariencia de cara al usuario final. Los trabajos definidos en ella son:

- Dar una movilidad mayor de los personajes que aparecen en la historia, mejorando la oscilación continua que caracteriza su movimiento en esta versión. Así poder dar a los personajes distintos tipos de movimientos, no sólo quedarse en giros o caminar en línea recta, pudiendo realizar recorridos circulares, movimientos de cabeza, etc.
- Incrementar las prestaciones gráficas en lo que a eficiencia se refiere. El motor gráfico jPCT soporta múltiples herramientas capaces de ajustar la representación al entorno renderizado. Sería oportuno llevar a cabo un tratamiento de dicho entorno de forma que se pudiesen utilizar dichas herramientas de forma correcta.

La segunda vertiente hace alusión a la interacción con los personajes y podría englobar las siguientes líneas de trabajo:

- Introducción de sonido (leer el texto) en las intervenciones de tipo test.

- Diseño de una interfaz de diálogo más potente, que permita al usuario interactuar con el personaje por medio de un lenguaje natural escrito.
- De manera complementaria al punto anterior se podría desarrollar un sistema de inteligencia artificial que permita recordar a los personajes anteriores diálogos con el protagonista. De esta forma, el devenir de la conversación se caracterizará por estos recuerdos así como por el perfil actual que caracteriza al usuario en su transcurso por la aventura. Esto supondría la inclusión de variables que almacenarían estados locales.
- Para los tests, usar un estándar de evaluación, como QTI (Question & Test Interoperabilits).
- Para los materiales didácticos, estudiar la posibilidad de incluir paquetes de contenidos educativos (formatos SCORM, IMS-LD) o conectar con plataformas educativas exteriores.

Finalmente, la tercera de las vertientes tiene que ver con el 'modo desarrollador' diseñado en esta plataforma.

- Tener la posibilidad de guardar el guión cuando el desarrollador quiera. En esta última versión sólo se puede guardar cada vez que el desarrollador finalice una fase, una de las mejoras puede ser poder guardarlo cuando la fase no esté completa.
- A su vez, tener la posibilidad de poder modificar una fase ya generada en el guión.
- Poder ubicar los focos de luz desde la aplicación donde el desarrollador desee.
- En caso de elegir un objeto 3D para colocar en el escenario, tener la posibilidad de cancelar dicha colocación en caso de cambiar de opinión.
- Poder reutilizar objetos introducidos en fases anteriores. En la aplicación actual se introducen siempre personajes nuevos a pesar de utilizar la misma imagen, una de las mejoras sería preguntar si se quiere reutilizar un personaje que tiene el mismo archivo introducido en una fase anterior o por el contrario se quiere añadir uno nuevo y así poder darle una escala nueva (o tener la posibilidad de modificar la escala reutilizando un personaje).

ANEXO I

A. DTD ‘juego.dtd’

DTD que valida el documento XML que define el guión de un juego.

```
<!ELEMENT Juego (Temporizador?,Pantalla?,Escenario,Protagonista,Personajes?,Ascensores?,Puertas?,Dialogos?,Fases?)>
<!ELEMENT Temporizador EMPTY>
<!ATTLIST Temporizador
		juego CDATA #IMPLIED
		teclado CDATA #IMPLIED
		representar CDATA #IMPLIED
		protagonista CDATA #IMPLIED
		personaje CDATA #IMPLIED
		ascensor CDATA #IMPLIED
		dialogo CDATA #IMPLIED
>
<!ELEMENT Pantalla EMPTY>
<!ATTLIST Pantalla
		titulo CDATA #IMPLIED
		anchopantalla CDATA #IMPLIED
		altopantalla CDATA #IMPLIED
		redimension (si|no) #IMPLIED
		collide_offset CDATA #IMPLIED
		tune_for_outdoor (si|no) #IMPLIED
		maximo_polys_visibles CDATA #IMPLIED
		renderizacion (software|hardware) #IMPLIED
		glmipmap (si|no) #IMPLIED
		gltrilinear (si|no) #IMPLIED
		glcolordepth CDATA #IMPLIED
		glzbufferdepth CDATA #IMPLIED
		glfixedblitting (si|no) #IMPLIED
>
<!ELEMENT Escenario (Mundo?,Entorno)>
		<!ELEMENT Mundo (Color_Luz?,Foco_Luz*)>
		<!ATTLIST Mundo
				activada_caida (si|no) #IMPLIED
				caida_de_luz CDATA #IMPLIED
				cobertura_luz CDATA #IMPLIED
		>
				<!ELEMENT Color_Luz EMPTY>
				<!ATTLIST Color_Luz
						r_luz_ambiente CDATA #REQUIRED
						g_luz_ambiente CDATA #REQUIRED
						b_luz_ambiente CDATA #REQUIRED
				>
				<!ELEMENT Foco_Luz EMPTY>
				<!ATTLIST Foco_Luz
						componente_rojo CDATA #REQUIRED
						componente_verde CDATA #REQUIRED
						componente_azul CDATA #REQUIRED
						posicion_x CDATA #REQUIRED
						posicion_y CDATA #REQUIRED
						posicion_z CDATA #REQUIRED
				>
		<!ELEMENT Entorno EMPTY>
		<!ATTLIST Entorno
				archivo_3ds CDATA #REQUIRED
				directorio_texturas CDATA #REQUIRED
				escala CDATA #REQUIRED
		>
<!ELEMENT Protagonista EMPTY>
<!ATTLIST Protagonista
		posicion_x CDATA #REQUIRED
		posicion_y CDATA #REQUIRED
		posicion_z CDATA #REQUIRED
		rotacion_y CDATA #IMPLIED
		altura CDATA #REQUIRED
		grosor CDATA #REQUIRED
		velocidad_movimiento CDATA #IMPLIED
		velocidad_giro CDATA #IMPLIED
		velocidad_caida CDATA #IMPLIED
>
<!ELEMENT Personajes (Personaje+)>
		<!ELEMENT Personaje EMPTY>
		<!ATTLIST Personaje
				id CDATA #REQUIRED
				archivo_3ds CDATA #REQUIRED
				directorio_texturas CDATA #REQUIRED
				escala CDATA #REQUIRED
				posicion_x CDATA #REQUIRED
				posicion_y CDATA #REQUIRED
				posicion_z CDATA #REQUIRED
				rotacion_x CDATA #IMPLIED
				rotacion_y CDATA #IMPLIED
				rotacion_z CDATA #IMPLIED
				movimiento (si|no) #IMPLIED
```

```
                velocidad CDATA #IMPLIED
                eje (x|z) #IMPLIED
                posFinal CDATA #IMPLIED
        >
<!ELEMENT Ascensores (Ascensor+)>
        <!ELEMENT Ascensor EMPTY>
        <!ATTLIST Ascensor
                id CDATA #REQUIRED
                archivo_3ds CDATA #REQUIRED
                directorio_texturas CDATA #REQUIRED
                escala CDATA #REQUIRED
                modo_funcionamiento (modo_alternativo|modo_movimiento|modo_planta1|modo_planta2) #IMPLIED
                posicion_x CDATA #REQUIRED
                posicion_y CDATA #REQUIRED
                posicion_z CDATA #REQUIRED
                posicion_planta1 CDATA #REQUIRED
                posicion_planta2 CDATA #REQUIRED
                velocidad CDATA #REQUIRED
                rotacion_x CDATA #IMPLIED
                rotacion_y CDATA #IMPLIED
                rotacion_z CDATA #IMPLIED
        >
<!ELEMENT Puertas (Puerta+)>
        <!ELEMENT Puerta EMPTY>
        <!ATTLIST Puerta
                id CDATA #REQUIRED
                archivo_3ds CDATA #REQUIRED
                directorio_texturas CDATA #REQUIRED
                escala CDATA #REQUIRED
                posicion_x CDATA #REQUIRED
                posicion_y CDATA #REQUIRED
                posicion_z CDATA #REQUIRED
                rotacion_x CDATA #IMPLIED
                rotacion_y CDATA #IMPLIED
                rotacion_z CDATA #IMPLIED
        >
<!ELEMENT Dialogos (Dialogo+)>
        <!ELEMENT Dialogo
((Intervencion_Texto|Intervencion_Label|Intervencion_Opcion|Intervencion_Test|Intervencion_Final|Intervencion_Tema|Intervecion_Comun), PartesComunes?)
>
        <!ATTLIST Dialogo
                id CDATA #REQUIRED
        >
                <!ELEMENT Intervencion_Texto
(Texto_Area,(Intervencion_Texto|Intervencion_Label|Intervencion_Opcion|Intervencion_Test|Intervencion_Final|Intervencion_Tema|Intervencion_Comun))>
                        <!ATTLIST Intervencion_Texto
                                sonido CDATA #IMPLIED>
                        <!ELEMENT Texto_Area (#PCDATA)>

                <!ELEMENT Intervencion_Opcion (Texto_Principal,Opcion+)>
                        <!ELEMENT Texto_Principal (#PCDATA)>
                        <!ELEMENT Opcion
(Texto_Opcion,(Intervencion_Texto|Intervencion_Label|Intervencion_Opcion|Intervencion_Test|Intervencion_Final|Intervencion_Tema|Intervencion_Comun))>
                                <!ELEMENT Texto_Opcion (#PCDATA)>
                        <!ATTLIST Intervencion_Opcion
                                sonido CDATA #IMPLIED>

                <!ELEMENT Intervencion_Test (Suspenso,Aprobado)>
                <!ATTLIST Intervencion_Test
                        numero_preguntas CDATA #REQUIRED
                        nota_suspenso           CDATA #REQUIRED
                        archivo_test CDATA #REQUIRED
                        sonido CDATA #IMPLIED
                >
                        <!ELEMENT Suspenso
(Intervencion_Texto|Intervencion_Label|Intervencion_Opcion|Intervencion_Test|Intervencion_Final|Intervencion_Tema|Intervencion_Comun)>
                        <!ATTLIST Suspenso
                                sonido CDATA #IMPLIED>
                        <!ELEMENT Aprobado
(Intervencion_Texto|Intervencion_Label|Intervencion_Opcion|Intervencion_Test|Intervencion_Final|Intervencion_Tema|Intervencion_Comun)>
                        <!ATTLIST Aprobado
                                sonido CDATA #IMPLIED>

                <!ELEMENT Intervencion_Final EMPTY>
                <!ATTLIST Intervencion_Final siguiente_fase CDATA #REQUIRED>

                <!ELEMENT Intervencion_Tema
(Intervencion_Texto|Intervencion_Label|Intervencion_Opcion|Intervencion_Test|Intervencion_Final|Intervencion_Tema|Intervencion_Comun)>
                <!ATTLIST Intervencion_Tema
                        archivo_tema CDATA #REQUIRED
                        tema CDATA #REQUIRED
                        sonido CDATA #IMPLIED
                >
                <!ELEMENT Intervencion_Comun EMPTY>
                <!ATTLIST Intervencion_Comun idComun CDATA #REQUIRED>

        <!ELEMENT PartesComunes (ParteComun+)>
                <!ELEMENT ParteComun
(Intervencion_Texto|Intervencion_Label|Intervencion_Opcion|Intervencion_Test|Intervencion_Final|Intervencion_Tema|Intervecion_Comun)>
                <!ATTLIST ParteComun idComun CDATA #REQUIRED>

<!ELEMENT Fases (Fase+)>
<!ATTLIST Fases fase_inicial CDATA #REQUIRED>
```

```
<!ELEMENT Fase (Personaje_Activo*,Ascensor_Activo*,Puerta_Activa*) >
<!ATTLIST Fase id_fase CDATA #REQUIRED>
        <!ELEMENT Personaje_Activo EMPTY>
        <!ATTLIST Personaje_Activo
                id CDATA #REQUIRED
                dialogo_id CDATA #IMPLIED
                posicionX CDATA #IMPLIED
                posicionY CDATA #IMPLIED
                posicionZ CDATA #IMPLIED
                rotacionY CDATA #IMPLIED
        >
        <!ELEMENT Ascensor_Activo EMPTY>
        <!ATTLIST Ascensor_Activo id CDATA #REQUIRED            >
        <!ELEMENT Puerta_Activa EMPTY>
        <!ATTLIST Puerta_Activa
                id CDATA #REQUIRED
                posicionX CDATA #IMPLIED
                posicionY CDATA #IMPLIED
                posicionZ CDATA #IMPLIED
                rotacionY CDATA #IMPLIED
        >
```

B. DTD 'curso.dtd'

DTD que valida el documento XML de contenidos didácticos

```
<!ELEMENT Curso (Tema+)>
<!ELEMENT Tema (#PCDATA)>
<!ATTLIST Tema numero_tema CDATA #REQUIRED>
```

C. DTD 'test.dtd'

DTD que valida los documentos XML que implementan las baterías de preguntas tipo test multi-respuesta.

```
<!ELEMENT Preguntas (Pregunta*)>
        <!ELEMENT Pregunta (Enunciado,Respuesta+)>
                <!ELEMENT Enunciado (#PCDATA)>
                <!ELEMENT Respuesta (#PCDATA)>
                <!ATTLIST Respuesta solucion (si|no) #REQUIRED>
```

ANEXO II: Manual del desarrollador

Este manual trata de guiar al desarrollador por cada uno de los pasos que debe recorrer a la hora de diseñar un nuevo guión mínimo de juego. En esta guía se exponen todas las definiciones que deben aparecer en el guión básico que utilizará la aplicación para la edición. Un guión que se utilizará para editar, también puede soportar otros elementos generados anteriormente en el modo edición (como fases, diálogos, personajes, puertas y ascensores).

Creación del guión del juego.

1. Adquirir los modelos de todos los componentes que intervendrán en el guión del juego. Para ello se podrán diseñar manualmente mediante el software '3D Studio' o bien descargar de páginas Web especializadas en ello.

   Una vez contamos con los modelos 3DS, se ubicarán en los directorios recomendados de modo que queden totalmente ordenados facilitando su acceso desde el documento XML una vez se haga referencia a ellos.

2. Definir la cabecera del documento XML, en la que se inidica la DTD utilizada para validar el contenido del mismo.

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE Juego SYSTEM "..\..\..\xml\juego.dtd">
…….
```

   En la definición de esta cabecera se tendrá en cuenta la ubicación del archivo que define la DTD, procediéndose a realizar un direccionamiento relativo que permita estandarizar el comportamiento del documento XML.

3. Definir el tiempo entre actuaciones para cada uno de los hilos. Este elemento es optativo, al igual que cada uno de los atributos que lo caracterizan, por ello este paso puede ser omitido, configurándose los valores por defecto para dichos atributos.

```
……
<Temporizador ascensor="30" dialogo="175" juego="180" personaje="80" protagonista="25" representar="25"
teclado="80"/>
……
```

4. Definir los parámetros que caracterizan el entorno gráfico de la aplicación (ver apartado 4.2.1). Este elemento, a igual que muchos de sus atributos, es optativo. Debido a ello, se puede omitir este paso, configurándose los valores por defecto asignados a cada una de las variables.

```
……
<Pantalla altopantalla="384" anchopantalla="512" collide_offset="250" glcolordepth="16" glfixedblitting="si"
glmipmap="si" gltrilinear="si" glzbufferdepth="16" maximo_polys_visibles="30000" redimension="no"
renderizacion="hardware" titulo="L &amp;amp; X Compilator" tune_for_outdoor="si"/>
……
```

5. Definir el entorno virtual donde tendrá lugar el juego. Para ello se especifica el modelo utilizado para diseñar la escena así como todos los focos de luz presentes en ella.

```
……
<Escenario>
   <Mundo activada_caida="si" caida_de_luz="100" cobertura_luz="300">
      <Color_Luz g_luz_ambiente="15" b_luz_ambiente="15" r_luz_ambiente="10"/>
      <Foco_Luz componente_rojo="25" componente_verde="25" componente_azul="25" posicion_x="850" posicion_y="-
150" posicion_z="-430"/>
```

```
      <Foco_Luz componente_rojo="25" componente_verde="25" componente_azul="25" posicion_x="850" posicion_y="-
150" posicion_z="-600"/>
      ……
   </Mundo>
   <Entorno directorio_texturas="juegos-Geografia-3ds-escenario" archivo_3ds="juegos-Geografia-3ds-escenario-ql.3ds"
escala="20f"/>
</Escenario>
……
```

6. Definición de las propiedades del protagonista. Para ello se crea el elemento obligado ‘Protagonista’ y se le asignan valor a los atributos que lo caracterizan.

```
<Protagonista posicion_x="620" posicion_y="-60" posicion_z="-600" grosor="6" altura="30" velocidad_caida="4f"
velocidad_movimiento="2.5f" velocidad_giro="0.05f" rotacion_y="3.14f"/>
```

ANEXO III: Manual del uso (Juegos diseñados para la plataforma)

Requisitos del equipo en el que se ejecuta la plataforma

- Espacio disponible en disco duro: 50 Mbytes.
- Memoria RAM mínima: 100 Mbytes.
  Memoria RAM recomendada: 512 Mbytes.
- Tarjeta gráfica compatible con OpenGL en caso de utilizar renderización por hardware.
- Software necesario: Versión Java 1.2 o posterior para una ejecución caracterizada por una renderización por software. Versión Java 1.4 o posterior si se utiliza renderización por hardware.

Instalación de la plataforma

Se lleva a cabo mediante la descompresión del archivo 'instalacion.zip' en cualquier directorio del equipo.

Carga de un juego

Para llevar a cabo este cometido será necesario copiar la carpeta, en cuyo contenido se encuentre el juego deseado, dentro del directorio 'juegos' del árbol creado tras la instalación de la plataforma. Cada uno de estos módulos será cargado de forma independiente pudiendo ejecutar cualquiera de ellos en un determinado momento.

Figura 75: Árbol de directorios

Ejecución

Para ejecutar un juego dentro de la plataforma será necesario introducir la siguiente línea de comandos.

```
java -Xmx400m -Djava.library.path=lib\lwjgl-1.0b3\ -classpath lib\lwjgl-1.0b3\lwjgl.jar;
lib\jpct\jpct.jar;.;lib\jdom\jdom.jar;lib\jdom\xerces.jar Juego  [–juego || -edicion]
```

En ella se especificará la cantidad de memoria RAM límite que podrá ser utilizada por la máquina virtual de java (-Xmx). En el ejemplo expuesto se está utilizando un límite de 400 Mbytes.

Posteriormente se hace referencia a todas las librerías utilizadas por nuestra aplicación y ubicadas en el directorio 'lib' del árbol creado tras la instalación de la plataforma.

En caso de omitir las opcines, la aplicación te preguntará si quieres editar o jugar y posteriormente te solicitará abrir el guión XML. Por el contrario, si introducimos la opción '-juego' la aplicación sabrá que lo que queremos es jugar, mientras que si le indicamos la opción '-edicion' pasaremos a editar el guión del juego.

Utilización del juego

La iteración del usuario final con el juego sintetizado por la plataforma seguirá las siguientes pautas:

- El control de movimientos del protagonista se realiza por medio de los cursores. Dicho actor se podrá mover hacia delante o hacia atrás siempre que pulse las teclas ↑ o ↓ respectivamente. De la misma forma, podrá girar a la izquierda o derecha siempre que pulse las teclas ← o →.
- Para llevar a cabo una conversación con alguno de los personajes, se colisionará con ellos. Esto provocará el evento necesario para lanzar la ventana emergente que implementa el diálogo correspondiente.

  Tras cerrar la ventana, se inhibirá la posibilidad de colisionar con dicho personaje durante un periodo de 5 segundos, tiempo que utilizará el usuario para alejarse y no volver a colisionar con él.
- Si pulsamos la tecla 's', aparecerá una pantalla preguntándonos si queremos sonido en el juego.
- Finalmente, en el caso de querer acabar con la ejecución de la aplicación, se tendrá que pulsar la tecla ESC desde la ventana principal.

Utilización del modo edición

Estando en el modo edición tenemos las siguientes pautas:

- Si estamos intentando mover un objeto para ubicarlo en el escenario utilizaremos las teclas 'k' o 'i' para moverlo sobre el eje Z, 'j' o 'l' para moverlo sobre el eje X, 'Re Pág' o 'Av Pág' para moverlo sobre el eje Y.
- Si lo que queremos es girarlo sobre alguno de sus ejes utilizaremos 'g' para girarlo sobre el eje Y, 'x' para girarlo sobre el eje X, 'z' para el eje Z.
- Para cambiar la escala de los objetos utilizaremos las teclas 'a' para aumentarla y 's' para decrementarla.
- Para fijar las coordenadas de los objetos (o protagonista en el caso final) utilizaremos la tecla 'c'.
- Finalmente para mover la cámara y poder seguir recorriendo el escenario visualmente se utilizarán los cursores. Para movernos hacia delante o hacia atrás se pulsarán las teclas ↑ o ↓ respectivamente. De la misma forma, podremos girar a la izquierda o derecha siempre que pulse las teclas ← o →.

ANEXO IV: Documentación de las clases Java

Class Audio

```
java.lang.Object
  └ Audio
```

All Implemented Interfaces:
java.lang.Runnable, java.util.EventListener, javax.sound.midi.MetaEventListener, javax.sound.sampled.LineListener

---

```
public class Audio
extends java.lang.Object
implements java.lang.Runnable, javax.sound.sampled.LineListener, javax.sound.midi.MetaEventListener
```

Clase con la que damos sonido a la aplicación

Constructor Summary
`Audio(java.lang.String fich)` Constructor de la clase

Method Summary	
`void`	`close()` Metodo para cerrar el audio
`boolean`	`getAbierto()` Metodo por el que sabemos si un sonido se esta reproduciendo
`boolean`	`loadSound(java.lang.Object object)` Metodo para cargar el archivo
`void`	`meta(javax.sound.midi.MetaMessage message)` Metodo por el que sabemos que se ha llegado al final del sonido
`void`	`open()` Método para abrir el audio
`void`	`playSound()` Metodo para reproducir el sonido
`void`	`reproducirUnaVez(boolean num)` Metodo por el que indicamos si se debe reproducir una vez el sonido o infinitas veces
`void`	`run()` Metodo utilizado cuando se arranca el hilo
`void`	`start()` Metodo para arrancar el sonido
`void`	`stop()` Metodo para cerrar el sonido
`void`	`update(javax.sound.sampled.LineEvent event)` Cabecera de metodo obligatorio de la interfaz

Methods inherited from class java.lang.Object
`clone, equals, finalize, getClass, hashCode, notify, notifyAll, toString, wait, wait, wait`

Constructor Detail

Audio

```
public Audio(java.lang.String fich)
```

Constructor de la clase

Parameters:
`fich` - es el fichero de audio, comprueba si tiene la extensión adecuada para poder reproducirlo

Method Detail

getAbierto

```
public boolean getAbierto()
```

Metodo por el que sabemos si un sonido se esta reproduciendo

Returns:
true si se esta reproduciendo, false en caso contrario

---

open

```
public void open()
```

Método para abrir el audio

---

close

`public void close()`

Metodo para cerrar el audio

---

loadSound

`public boolean loadSound(java.lang.Object object)`

Metodo para cargar el archivo

Parameters:

`object,` - objeto a cargar

---

playSound

`public void playSound()`

Metodo para reproducir el sonido

---

update

`public void update(javax.sound.sampled.LineEvent event)`

Cabecera de metodo obligatorio de la interfaz

Specified by:

`update` in interface `javax.sound.sampled.LineListener`

---

meta

`public void meta(javax.sound.midi.MetaMessage message)`

Metodo por el que sabemos que se ha llegado al final del sonido

Specified by:

`meta` in interface `javax.sound.midi.MetaEventListener`

---

start

`public void start()`

Metodo para arrancar el sonido

---

stop

`public void stop()`

Metodo para cerrar el sonido

---

reproducirUnaVez

`public void reproducirUnaVez(boolean num)`

Metodo por el que indicamos si se debe reproducir una vez el sonido o infinitas veces

Parameters:

`true` - si se reproduce solo una vez, false para infinitas

---

run

`public void run()`

Metodo utilizado cuando se arranca el hilo

Specified by:

`run` in interface `java.lang.Runnable`

Class Boton

`java.lang.Object`
  └ GUIComponent
    └ **Boton**

`public class Boton`
`extends GUIComponent`

Clase con la cual podemos crear un botón que sea capaz de utilizar la aplicación

---

Field Summary

Fields inherited from class GUIComponent
`caracter`, `visible`

Constructor Summary
`Boton``(int xpos, int ypos, int xdim, int ydim)` Constructor de la clase

Method Summary	
void	**draw**(com.threed.jpct.FrameBuffer buffer) Método por el cuál podemos pintar en botón en la aplicación
boolean	**evaluateInput**(MouseMapper mouse, com.threed.jpct.util.KeyMapper keyMapper) Evalúa si se ha producido un evento
boolean	**getActivo**() Método por el cual sabemos si hay un botón activado
java.lang.String	**getEtiqueta**() Método con el que obtenemos la etiqueta del botón
int	**getId**() Método por el cual obtenemos el id del botón
java.lang.String	**getPosicion**() Método con el cuál obtenemos la posición del botón
void	**setEtiqueta**(java.lang.String etiqueta) Método con el que modificamos la etiqueta del botón
void	**setId**(int id) Método con el cuál modificamos el id del botón
void	**setListener**(GUIListener bl) Método por el cual modificamos el objeto que maneja los eventos que se pueden producir en el botón
void	**setPosicion**(java.lang.String posicion) Método por el cual indicamos la posición (indice) del botón.
void	**setTextoVisible**(boolean hide) Método con el que indicamos si la etiqueta del botón es visible
void	**setX**(int xpos) Metodo por el que modificamos la posicion X
void	**setY**(int ypos) Metodo por el que modificamos la posicion Y

Methods inherited from class GUIComponent
add, getParent, getParentX, getParentY, getX, getY, isVisible, remove, setCaracter, setVisible, setXFigura, setYFigura

Methods inherited from class java.lang.Object
clone, equals, finalize, getClass, hashCode, notify, notifyAll, toString, wait, wait, wait

Constructor Detail

Boton

```
public Boton(int xpos,
             int ypos,
             int xdim,
             int ydim)
```

Constructor de la clase

Parameters:

`xpos` - es la posicion x de la esquina superior izquierda del botón
`ypos` - es la posición y de la esquina superior izquierda del botón
`xdim` - es el largo del botón
`ydim` - es el ancho del botón

Method Detail

setX

```
public void setX(int xpos)
```

Metodo por el que modificamos la posicion X

Overrides:

setX in class GUIComponent

Parameters:

`xpos` - es la nueva posicion X

---

setY

```
public void setY(int ypos)
```

Metodo por el que modificamos la posicion Y

Overrides:

setY in class GUIComponent

Parameters:

`ypos` - es la nueva posicion Y

---

setTextoVisible

```
public void setTextoVisible(boolean hide)
```

Método con el que indicamos si la etiqueta del botón es visible

Parameters:

`hide` - indica si la etiqueta está oculta: true el texto del botón no se ve, false en caso contrario

---

setEtiqueta

```
public void setEtiqueta(java.lang.String etiqueta)
```

Método con el que modificamos la etiqueta del botón

Parameters:

`etiqueta` - es el nombre que contendrá el botón

---

getEtiqueta

```
public java.lang.String getEtiqueta()
```

Método con el que obtenemos la etiqueta del botón

Returns:

etiqueta es el nombre de la etiqueta del botón

---

getActivo

```
public boolean getActivo()
```

Método por el cual sabemos si hay un botón activado

Returns:

true si está activado (*), false en caso contrario

---

setPosicion

```
public void setPosicion(java.lang.String posicion)
```

Método por el cual indicamos la posición (indice) del botón. Es utilizado para botones de tipo verificación, para saber qué botón está activado (verificado)

Parameters:

`posicion` - es la posicion del botón

---

getPosicion

```
public java.lang.String getPosicion()
```

Método con el cuál obtenemos la posición del botón

Returns:

posicion del botón

---

setId

```
public void setId(int id)
```

Método con el cuál modificamos el id del botón

Parameters:

`id` - nuevo id

---

getId

```
public int getId()
```

Método por el cual obtenemos el id del botón

Returns:

id

---

setListener

```
public void setListener(GUIListener bl)
```

Método por el cual modificamos el objeto que maneja los eventos que se pueden producir en el botón

Parameters:

`bl` - es el nuevo manejador

---

evaluateInput

```
public boolean evaluateInput(MouseMapper mouse,
                             com.threed.jpct.util.KeyMapper keyMapper)
```

Evalúa si se ha producido un evento

Overrides:

`evaluateInput` in class `GUIComponent`

Parameters:

`mouse` - es el objeto manejador del raton

`keyMapper` - es el objeto manejador del teclado

Returns:

true si hay un evento nuevo, false en caso contrario

---

draw

```
public void draw(com.threed.jpct.FrameBuffer buffer)
```

Método por el cuál podemos pintar en botón en la aplicación

Overrides:

`draw` in class `GUIComponent`

Parameters:

`buffer` - donde pintamos

Class CampoTexto

```
java.lang.Object
  └ GUIComponent
      └ CampoTexto
```

```
public class CampoTexto
extends GUIComponent
```

Clase con la que podemos introducir datos en la aplicación

Field Summary

Fields inherited from class GUIComponent
caracter, visible

Constructor Summary
CampoTexto(int xpos, int ypos, int xdim, int ydim) Constructor de la clase

Method Summary	
void	**bajar**() Metodo utilizado para desplazar el texto en una ventana cuando pulsamos en el boton de bajar para ver lo que habrá posteriormente a lo que se muestra por pantalla (muestra lo que hay una línea después)
void	**borrar**() Método con el que limpiamos el campo de texto
void	**draw**(com.threed.jpct.FrameBuffer buffer) Método por el cual dibujamos el campo de texto
boolean	**evaluateInput**(MouseMapper mouse, com.threed.jpct.util.KeyMapper mapper) Método que evalúa si se ha producido algún evento
java.lang.String	**getTexto**() Método que nos devuelve el contenido del campo de texto
void	**setTexto**(java.lang.String txt) Método que modifica el contenido del campo de texto
void	**setX**(int xp) Metodo por el que se modifica la posicion X
void	**setXDim**(int xs) Metodo por el que se modifica el largo del campo
void	**setY**(int yp) Metodo por el que se modifica la posicion Y
void	**setYDim**(int ys) Metodo por el que se modifica el ancho del campo
void	**subir**() Metodo utilizado para desplazar el texto en una ventana cuando pulsamos en el boton de subir para ver lo que había anteriormente (ve lo que hay una línea antes)

Methods inherited from class GUIComponent
add, getParent, getParentX, getParentY, getX, getY, isVisible, remove, setCaracter, setVisible, setXFigura, setYFigura

Methods inherited from class java.lang.Object
clone, equals, finalize, getClass, hashCode, notify, notifyAll, toString, wait, wait, wait

Constructor Detail

CampoTexto

```
public CampoTexto(int xpos,
                  int ypos,
                  int xdim,
                  int ydim)
```

Constructor de la clase

Parameters:

`xpos` - es la posicion x de la esquina superior izquierda
`ypos` - es la posicion y de la esquina superior izquierda
`xdim` - es el tamaño de la dirección x
`ydim` - es el tamaño de la dirección y

Method Detail

getTexto

```
public java.lang.String getTexto()
```

Método que nos devuelve el contenido del campo de texto

Returns:

contenido del campo de texto

setY

`public void setY(int yp)`

Metodo por el que se modifica la posicion Y

Overrides:

`setY` in class `GUIComponent`

Parameters:

`yp` - es la nueva coordenada Y

---

setX

`public void setX(int xp)`

Metodo por el que se modifica la posicion X

Overrides:

`setX` in class `GUIComponent`

Parameters:

`xp` - es la nueva coordenada X

---

setXDim

`public void setXDim(int xs)`

Metodo por el que se modifica el largo del campo

Parameters:

`xs` - es el nuevo largo

---

setYDim

`public void setYDim(int ys)`

Metodo por el que se modifica el ancho del campo

Parameters:

`ys` - es el nuevo ancho

---

setTexto

`public void setTexto(java.lang.String txt)`

Método que modifica el contenido del campo de texto

Parameters:

`txt` - es el nuevo contenido del campo de texto

---

borrar

`public void borrar()`

Método con el que limpiamos el campo de texto

---

evaluateInput

```
public boolean evaluateInput(MouseMapper mouse,
                             com.threed.jpct.util.KeyMapper mapper)
```

Método que evalúa si se ha producido algún evento

Overrides:

`evaluateInput` in class `GUIComponent`

Parameters:

`mouse` - comprueba el manejo del ratón

`mapper` - comprueta el manejo del teclado

Returns:

true si hay algún evento, false en caso contrario

---

bajar

`public void bajar()`

Metodo utilizado para desplazar el texto en una ventana cuando pulsamos en el boton de bajar para ver lo que habrá posteriormente a lo que se muestra por pantalla (muestra lo que hay una línea después)

---

subir

`public void subir()`

Metodo utilizado para desplazar el texto en una ventana cuando pulsamos en el boton de subir para ver lo que había anteriormente (ve lo que hay una línea antes)

---

draw

`public void draw(com.threed.jpct.FrameBuffer buffer)`

Método por el cual dibujamos el campo de texto

Overrides:

`draw` in class `GUIComponent`

Parameters:

`buffer` - donde dibujamos

Class Componente

```
java.lang.Object
  └ Componente
```

Direct Known Subclasses:

Ascensor, Entorno, MenuObjetos, Personaje, Puerta

`public abstract class Componente`

extends java.lang.Object

Clase abstracta que representa un componente generico del escenario, tal como el entorno, los personajes, ascensores y otros elementos que vienen especificados por la clase Puerta. Esta clase Implementará metodos genericos a todos estos componentes, ademas de contener atributos importantes en la definición de cada una de sus subclases.

Since:
.JDK 1.4

See Also:
Ascensor, Entorno, Personaje, Puerta

Field Summary	
boolean	**activado** Variable booleana que indica si el componente se encuentra activado o no.
int	**id_componente** Identificador del componente del que se servirán los metodos para acceder a un objeto en concreto.
com.threed.jpct.Object3D	**objeto** Objeto 3D sobre el que se opera, y que contituye la base del componente
com.threed.jpct.SimpleVector	**posfija** Variable que alamcena el valor de la posicion inicial del componente una vez llamada al método posicionarComponente del objeto correspondiente.
float	**rotfija** Variable que alamcena el valor de la rotacion inicial del componente una vez llamada al método posicionarComponente del objeto correspondiente.

Constructor Summary
Componente() Constructor que crea un objeto genérico.
Componente(com.threed.jpct.Object3D obj) Constructor que crea el objeto principal a partir de un objeto dado (cilindro, cono,...).

Method Summary	
com.threed.jpct.Object3D	**constructorComponente**(java.lang.String path3ds) Metodo encargado de la construccion del objeto en sí, abriendo el archivo 3ds, uniendo todas las partes de las que se compone, y configurando parametros importantes, tales como los modos de colisión.
void	**constructorComponente**(java.lang.String path3ds, float escala) Metodo encargado de la construccion del objeto en sí, abriendo el archivo 3ds, uniendo todas las partes de las que se compone, y configurando parametros importantes, tales como los modos de colisión.
boolean	**getActivacion**() Devuelve si el componente se encuentra activo o no
float	**getAngulo**() Método con el que obtenemos el ángulo con respecto al eje Y
float	**getAnguloX**() Método con el que obtenemos el angulo con respecto al eje x
float	**getAnguloZ**() Método con el que obtenemos el angulo con respecto al eje z
java.lang.String	**getArchivo3D**() Metodo con el que obtenemos la ruta del archivo 3D
java.lang.String	**getDirectorioTexturas**() Método con el que obtenemos la ruta del directorio de texturas
float	**getEscala**() Método con el que obtenemos la escala del objeto 3D
int	**getIdComponente**() Devuelve el valor del identificador del componente
com.threed.jpct.Object3D	**getObjeto**() Método con el que obtenemos el objeto 3D
com.threed.jpct.SimpleVector	**getOrigen**() Método con el que obtenemos la posición del objeto 3D
abstract void	**perfilarComponente**() Metodo abstracto que complementará los objetos determinados con las especificaciones concretas para su construcción.
com.threed.jpct.TextureManager	**recogerTexturas**(java.lang.String ptexturas) Metodo encargado de devolver las texturas ubicadas en el directorio pasado como parámetro en un manejador de texturas.
void	**reposicionarComponente**(com.threed.jpct.SimpleVector posicion_ini, float rotacion_ini) Metodo que reposiciona el componente, ubicandolo en la nueva posición indicada y con una rotacion relativa con respecto a la rotacion inicial indicada en el metodo posicionarComponente.
void	**rotarX**(float angulo) Rota el componente un angulo espeficiado en el eje X

void	**rotarZ**(float angulo) Rota el componente un angulo espeficiado en el eje Z
void	**setAngulo**(float angulo) Método con el que modificamos el ángulo con respecto al eje Y
void	**setAnguloX**(float angulo) Método con el que modificamos el ángulo con respecto al eje X
void	**setAnguloZ**(float angulo) Método con el que modificamos el ángulo con respecto al eje Z
void	**setArchivo3D**(java.lang.String archivo3D) Método que modifica la ruta del archivo 3D
void	**setDirectorioTexturas**(java.lang.String directorioTexturas) Método que modifica la ruta del archivo de texturas
void	**setEscala**(float escala) Método con el que modificamos la escala del objeto 3D
void	**setObjeto3D**(com.threed.jpct.Object3D obj) Método con el que modificamos el objeto 3D
void	**setOrigen**(com.threed.jpct.SimpleVector origen) Método con el que modificamos la posicion del objeto 3D

Methods inherited from class java.lang.Object

clone, equals, finalize, getClass, hashCode, notify, notifyAll, toString, wait, wait, wait

Field Detail

activado

public boolean **activado**

Variable booleana que indica si el componente se encuentra activado o no. En el caso de ser false, tal componente no realizará ninguna de sus actividades comunes.

See Also:
getActivacion

id_componente

public int **id_componente**

Identificador del componente del que se servirán los metodos para acceder a un objeto en concreto. Se trata de un entero con valores comprendidos entre 0 e infinito y nunca será negativo.

See Also:
getActivacion

objeto

public com.threed.jpct.Object3D **objeto**

Objeto 3D sobre el que se opera, y que contituye la base del componente

rotfija

public float **rotfija**

Variable que alamcena el valor de la rotacion inicial del componente una vez llamada al método posicionarComponente del objeto correspondiente. A partir de dicho valor se realizarán las rotaciones previas a traves del método reposicionarComponente.

See Also:
reposicionarComponente(com.threed.jpct.SimpleVector, float),
Ascensor.posicionarComponente(com.threed.jpct.SimpleVector, float),
Personaje.posicionarComponente(com.threed.jpct.SimpleVector, float),
Puerta.posicionarComponente(com.threed.jpct.SimpleVector, float)

posfija

public com.threed.jpct.SimpleVector **posfija**

Variable que alamcena el valor de la posicion inicial del componente una vez llamada al método posicionarComponente del objeto correspondiente. A partir de dicho valor se realizarán los reposicionamientos previos a traves del método reposicionarComponente.

See Also:
reposicionarComponente(com.threed.jpct.SimpleVector, float),
Ascensor.posicionarComponente(com.threed.jpct.SimpleVector, float),
Personaje.posicionarComponente(com.threed.jpct.SimpleVector, float),
Puerta.posicionarComponente(com.threed.jpct.SimpleVector, float)

Constructor Detail

Componente

public **Componente**()

Constructor que crea un objeto genérico.

Componente

public **Componente**(com.threed.jpct.Object3D obj)

Constructor que crea el objeto principal a partir de un objeto dado (cilindro, cono,...).

Parameters:
obj - Objeto a partir del cual se crea el objeto principal.

Method Detail

setArchivo3D

`public void setArchivo3D(java.lang.String archivo3D)`

Método que modifica la ruta del archivo 3D

Parameters:

`archivo3D` - es la nueva ruta

---

setDirectorioTexturas

`public void setDirectorioTexturas(java.lang.String directorioTexturas)`

Método que modifica la ruta del archivo de texturas

Parameters:

`directorioTexturas` - es la nueva ruta

---

getArchivo3D

`public java.lang.String getArchivo3D()`

Metodo con el que obtenemos la ruta del archivo 3D

Returns:

ruta 3D

---

getDirectorioTexturas

`public java.lang.String getDirectorioTexturas()`

Método con el que obtenemos la ruta del directorio de texturas

Returns:

ruta texturas

---

getObjeto

`public com.threed.jpct.Object3D getObjeto()`

Método con el que obtenemos el objeto 3D

Returns:

objeto 3D

---

setEscala

`public void setEscala(float escala)`

Método con el que modificamos la escala del objeto 3D

Parameters:

`escala` - nueva

---

getEscala

`public float getEscala()`

Método con el que obtenemos la escala del objeto 3D

Returns:

escala

---

reposicionarComponente

```
public void reposicionarComponente(com.threed.jpct.SimpleVector posicion_ini,
                                   float rotacion_ini)
```

Metodo que reposiciona el componente, ubicandolo en la nueva posición indicada y con una rotacion relativa con respecto a la rotacion inicial indicada en el metodo posicionarComponente.

Parameters:

`posicion_ini` - Posicion en la que ubicar el el componente

`rotacion_ini` - Rotacion relativa con respecto a la rotacion inicial expresada en radianes.

See Also:

`Ascensor.posicionarComponente(com.threed.jpct.SimpleVector, float)`,
`Personaje.posicionarComponente(com.threed.jpct.SimpleVector, float)`,
`Puerta.posicionarComponente(com.threed.jpct.SimpleVector, float)`

---

recogerTexturas

`public com.threed.jpct.TextureManager recogerTexturas(java.lang.String ptexturas)`

Metodo encargado de devolver las texturas ubicadas en el directorio pasado como parámetro en un manejador de texturas. Unicamente reconocerá como texturas archivos pasados con la extensión jpg.

Parameters:

`ptexturas` - Path del directorio donde se encuentran las texturas.

Returns:

Manejador de texturas con todas las texturas ubicadas en el directorio añadidas.

---

constructorComponente

```
public void constructorComponente(java.lang.String path3ds,
                                  float escala)
```

Metodo encargado de la construccion del objeto en sí, abriendo el archivo 3ds, uniendo todas las partes de las que se compone, y configurando parametros importantes, tales como los modos de colisión.

Parameters:

`path3ds` - Path del archivo 3ds del componente

`escala` - Escala del tamaño del componente

---

constructorComponente

`public com.threed.jpct.Object3D constructorComponente(java.lang.String path3ds)`

Metodo encargado de la construccion del objeto en sí, abriendo el archivo 3ds, uniendo todas las partes de las que se compone, y configurando parametros importantes, tales como los modos de colisión.

Parameters:
`path3ds` - Path del archivo 3ds del componente
Returns:
el objeto creado

---

getActivacion

`public boolean` **`getActivacion`**`()`

Devuelve si el componente se encuentra activo o no
Returns:
Boolean que especifica si esta o no activo.

---

getIdComponente

`public int` **`getIdComponente`**`()`

Devuelve el valor del identificador del componente
Returns:
Identificador del componente

---

rotarX

`public void` **`rotarX`**`(float angulo)`

Rota el componente un angulo espeficiado en el eje X
Parameters:
`angulo` - Angulo en radianes de rotacion

---

rotarZ

`public void` **`rotarZ`**`(float angulo)`

Rota el componente un angulo espeficiado en el eje Z
Parameters:
`angulo` - Angulo en radianes de rotacion

---

perfilarComponente

`public abstract void` **`perfilarComponente`**`()`

Metodo abstracto que complementará los objetos determinados con las especificaciones concretas para su construcción.

---

setOrigen

`public void` **`setOrigen`**`(com.threed.jpct.SimpleVector origen)`

Método con el que modificamos la posicion del objeto 3D
Parameters:
`origen` - es la nueva posicion

---

getOrigen

`public com.threed.jpct.SimpleVector` **`getOrigen`**`()`

Método con el que obtenemos la posición del objeto 3D
Returns:
posicion

---

setAngulo

`public void` **`setAngulo`**`(float angulo)`

Método con el que modificamos el ángulo con respecto al eje Y
Parameters:
`angulo` - es el nuevo angulo Y

---

getAngulo

`public float` **`getAngulo`**`()`

Método con el que obtenemos el ángulo con respecto al eje Y
Returns:
angulo Y

---

setAnguloX

`public void` **`setAnguloX`**`(float angulo)`

Método con el que modificamos el ángulo con respecto al eje X
Parameters:
`angulo` - es el nuevo angulo X

---

getAnguloX

`public float` **`getAnguloX`**`()`

Método con el que obtenemos el angulo con respecto al eje x
Returns:
angulo

---

setAnguloZ

`public void` **`setAnguloZ`**`(float angulo)`

Método con el que modificamos el ángulo con respecto al eje Z
Parameters:
`angulo` - es el nuevo angulo Z

---

getAnguloZ

`public float getAnguloZ()`

Método con el que obtenemos el angulo con respecto al eje z

Returns:

angulo

---

setObjeto3D

`public void setObjeto3D(com.threed.jpct.Object3D obj)`

Método con el que modificamos el objeto 3D

Parameters:

`obj` - es el nuevo objeto 3D

Class Conversacion

`java.lang.Object`
`└ Conversacion`

`public class Conversacion`
`extends java.lang.Object`

Clase que comprende atributos y metodos que caracterizan una conversación así como su estado a lo largo de un dialogo.

Since:

.JDK 1.4

See Also:

`Intervencion`, `Dialogo`

Field Summary	
`Intervencion`	**`actual`** Variable que guarda el valor de la intervención actual en una conversacion
`Intervencion[]`	**`hijos`** Lista de hijos de una intervencio.
`int`	**`num_test`** Numero de opciones ofrecidas como respuestas a un test.

Constructor Summary
`Conversacion``()` Constructor por defecto
`Conversacion``(Intervencion ini)` Construye una conversación en la que el nodo inicial es la intervencion ini y el numero de opciones en las preguntas de test viene fijado por num_test, no pudiendo ser diferente a lo largo de una misma conversacion.
`Conversacion``(Intervencion ini, Intervencion[] comun)` Construye una conversación en la que el nodo inicial es la intervencion ini y la parte comun que tiene un dialogo

Method Summary	
`void`	**`cerrarConversacion`**`()` Realiza el cierre de una conversacion reposicionando como actual la intervención inicial
`Intervencion[]`	**`getComun`**`()` Método por el que obtenemos la parte comun de un dialogo
`Intervencion`	**`pasoAHijo`**`(int i)` Realiza el paso a la intervención hijo especificada por parámetro.
`void`	**`setActual`**`(Intervencion actual)` Método por el que modificamos la intervencion actual de la conversacion
`void`	**`setComun`**`(Intervencion[] comun)` Método por el modificamos la parte comun de la conversacion
`void`	**`setConversacion`**`(Intervencion ini)` Metodo por el que modificamos la intervencion actual y el numero de opciones ofrecidas en un test

Methods inherited from class java.lang.Object
`clone, equals, finalize, getClass, hashCode, notify, notifyAll, toString, wait, wait, wait`

Field Detail

actual

`public Intervencion actual`

Variable que guarda el valor de la intervención actual en una conversacion

See Also:

`Intervencion`

---

hijos

`public Intervencion[] hijos`

Lista de hijos de una intervencio. Dichos hijos son intervenciones de diversos tipos de modo que las intervenciones de solo texto unicamente tendrán como hijo la intervencion siguiente, la intervenciones opcion tendrán tantos hijos como opciones posibles a elegir, y las intervenciones test tendrán dos hijos (Suspenso[posicion 0] y Aprobado[posicion 1]).

See Also:
`Intervencion`

---

num_test

`public int num_test`

Numero de opciones ofrecidas como respuestas a un test. Es comun en todas las intervenciones de una misma conversación, condición que se tratará para que se cumpla en todo momento, produciendo un fallo de compilación del juego en caso contrario.

Constructor Detail

Conversacion

`public Conversacion(Intervencion ini)`

Construye una conversación en la que el nodo inicial es la intervencion ini y el numero de opciones en las preguntas de test viene fijado por num_test, no pudiendo ser diferente a lo largo de una misma conversacion.

Parameters:
`ini` - Intervencion inicial
`num_test` - Numero de opciones ofrecidas como respuestas a un test.
See Also:
`Intervencion`, `num_test`

Conversacion

`public Conversacion()`

Constructor por defecto

Conversacion

`public Conversacion(Intervencion ini, Intervencion[] comun)`

Construye una conversación en la que el nodo inicial es la intervencion ini y la parte comun que tiene un dialogo

Parameters:
`ini` - Intervencion inicial
`comun` - Intervencion [] comun
See Also:
`Intervencion`

Method Detail

setConversacion

`public void setConversacion(Intervencion ini)`

Metodo por el que modificamos la intervencion actual y el numero de opciones ofrecidas en un test

Parameters:
`ini` - Intervencion inicial nueva
`num_test` - Numero de opciones ofrecidas como respuestas a un test.

---

setComun

`public void setComun(Intervencion[] comun)`

Método por el modificamos la parte comun de la conversacion

Parameters:
`comun` - Intervencion[] comun

---

setActual

`public void setActual(Intervencion actual)`

Método por el que modificamos la intervencion actual de la conversacion

Parameters:
`actual` - nuevo

---

getComun

`public Intervencion[] getComun()`

Método por el que obtenemos la parte comun de un dialogo

Returns:
comun

---

pasoAHijo

`public Intervencion pasoAHijo(int i)`

Realiza el paso a la intervención hijo especificada por parámetro.

Parameters:
`i` - Indice de la intervencion hijo a la que se pasa.
Returns:
Intervención actual una vez realizado el transpaso.
See Also:
`Intervencion`, `hijos`

---

cerrarConversacion

`public void cerrarConversacion()`

Realiza el cierre de una conversacion reposicionando como actual la intervención inicial

Class Cuestionario

```
java.lang.Object
  └ org.jdom.Content
      └ org.jdom.DocType
          └ Cuestionario
```

All Implemented Interfaces:
java.io.Serializable, java.lang.Cloneable

```
public class Cuestionario
extends org.jdom.DocType
```

Clase que se emplea para crear los documentos xml referente a los tests que utilizará la aplicación además si en el directorio donde se guardarán dichos documentos hay ficheros tests se cargarán en la vista de la aplicación

Since:
.JDK 1.4

See Also:
Docencia, Serialized Form

Field Summary

Fields inherited from class org.jdom.DocType

`elementName, internalSubset, publicID, systemID`

Fields inherited from class org.jdom.Content

`parent`

Constructor Summary

Cuestionario()	Constructor por defecto
Cuestionario(java.lang.String fichero)	Constructor de la clase
Cuestionario(java.util.Vector tests)	Constructor de la clase

Method Summary

void	**construirCuestionario**(org.jdom.Element raiz) Metodo por el cual tras pasar un fichero en el constructor podemos crear los objetos preguntas
org.jdom.DocType	**crearDocType**() Metodo por el cual le decimos al documento xml cual es el dtd que tiene que seguir
void	**crearFichero**() Metodo por el cual creamos el fichero xml con los tests utilizados por la aplicacion
java.lang.String	**getFichero**()
java.util.Vector	**getTests**() Metodo por el cual obtenemos todos los temas de la aplicacion
void	**setTests**(java.util.Vector tests) Metodo por el cual modificamos los temas

Methods inherited from class org.jdom.DocType

`getElementName, getInternalSubset, getPublicID, getSystemID, getValue, setElementName, setInternalSubset, setPublicID, setSystemID, toString`

Methods inherited from class org.jdom.Content

`clone, detach, equals, getDocument, getParent, getParentElement, hashCode, setParent`

Methods inherited from class java.lang.Object

`finalize, getClass, notify, notifyAll, wait, wait, wait`

Constructor Detail

Cuestionario

```
public Cuestionario(java.util.Vector tests)
```

Constructor de la clase

Parameters:
`tests` - es el vector que contiene los tests de la aplicacion

Cuestionario

`public Cuestionario()`

Constructor por defecto

Cuestionario

`public Cuestionario(java.lang.String fichero)`

Constructor de la clase

Parameters:

`fichero` - es el nombre del directorio donde puede haber tests que utilice la aplicacion para probar al usuario su conocimiento sobre el tema impartido

Method Detail

getFichero

`public java.lang.String getFichero()`

---

crearDocType

`public org.jdom.DocType crearDocType()`

Metodo por el cual le decimos al documento xml cual es el dtd que tiene que seguir

Returns:

doctype del dtd

---

setTests

`public void setTests(java.util.Vector tests)`

Metodo por el cual modificamos los temas

Parameters:

`temas` - son los nuevos temas del temario

---

getTests

`public java.util.Vector getTests()`

Metodo por el cual obtenemos todos los temas de la aplicacion

Returns:

temas

---

construirCuestionario

`public void construirCuestionario(org.jdom.Element raiz)`

Metodo por el cual tras pasar un fichero en el constructor podemos crear los objetos preguntas

Parameters:

`raiz` - es el elemento raiz del fichero xml que contiene las preguntas

---

crearFichero

`public void crearFichero()`

Metodo por el cual creamos el fichero xml con los tests utilizados por la aplicacion

Class Dialogo

```
java.lang.Object
  └ Dialogo
```

All Implemented Interfaces:

GUIListener, java.lang.Runnable

```
public class Dialogo
extends java.lang.Object
implements GUIListener, java.lang.Runnable
```

Since:

.JDK 1.4

See Also:

`Personaje`, `TransicionEstados`

Field Summary

`Conversacion`	**`conversacion`** Elemento conversación que caracteriza al objeto diálogo correspondiente
`int`	**`id_dialogo`** Identificador del dialogo
`int`	**`ind_preg`** Variable auxiliar que lleva la cuenta del numero de pregunta por el que nos encontramos en una bateria de estas de un test.
`TransicionEstados`	**`maquinaestados`** Objeto que engloba la maquina de estados principal del juego, y apartir del cual se podrá acceder a métodos para la transición entre estos,...
`static int`	**`MUESTREO`** Tiempo(ms) que duerme el hilo del objeto entre iteraciones consecutivas.
`int`	**`nota`** Variable que guarda el numero de fallos que se van teniendo a lo largo de la realización de un test te modo que si se llegase a un limite especificado se optaria por suspender este.

Representar	**representar** Donde se representará la ventana de diálogo.
boolean	**vivo** Variable que nos dirá si el hilo está vivo

Constructor Summary

Dialogo() Constructor por defecto
Dialogo(int iddial, Conversacion conversacion) Constructor de un diálogo caracterizado por un identificador concreto.

Method Summary

boolean	**activo**() Comprueba si el hilo sigue vivo
void	**areaTexto**() Metodo por el que crearemos una ventana de dialogo simple, es decir, en la que no haya interacción con el usuario, simplemente el personaje dirá su texto sin dé ninguna opción al jugador
void	**cincoOpciones**(Intervencion interv, boolean test, int indice) Metodo por el cual se crea una ventana con cinco opciones
void	**cuatroOpciones**(Intervencion interv, boolean test, int indice) Metodo por el cual se crea una ventana con cuatro opciones
void	**dosOpciones**(Intervencion interv, boolean test, int indice) Metodo por el cual se crea una ventana con dos opciones
void	**draw**(com.threed.jpct.FrameBuffer buffer) Dibuja la ventana de diálogo
void	**elementChanged**(java.lang.String label, java.lang.String data) Metodo encargado de gestionar los eventos producidos una vez se ha pulsado algun boton.
void	**evaluateInput**(MouseMapper mouse, com.threed.jpct.util.KeyMapper keyMapper) Comprueba si ha ocurrido algún evento en el diálogo
boolean	**isVisible**() Comprueba si la ventana es visible
void	**ocultar**() Oculta la ventana en caso de que sea visible
void	**opciones**(Intervencion interv, int num, boolean test, int indice) Metodo con el que crearemos opciones en la ventana de dialogo
void	**run**() Metodo que inicia el diálogo, y mientras vivo sea true refrescará la imagenes de la animación cada cierto tiempo impuesto por MUESTREO, produciendo así una animación.
void	**setMaquinaEstados**(TransicionEstados maquinaestados) Selecciónа la maquina de estados que caracteriza el Juego.
void	**setRepresentar**(Representar r) Metodo con el cual asignamos dónde se va a representar nuestro dialogo
void	**setSonido**(boolean sonido) Metodo por el que indicamos si hay que reproducir el sonido o no
void	**setTexto**(Etiqueta etiqueta, java.lang.String label) Modifica el texto que contiene la ventana de dialogo
void	**setVisible**() Metodo que hace que la ventana sea visible en el juego
void	**sonido**() Metodo por el que reproducimos el sonido de la intervencion actual
void	**tresOpciones**(Intervencion interv, boolean test, int indice) Metodo por el cual se crea una ventana con tres opciones

Methods inherited from class java.lang.Object

clone, equals, finalize, getClass, hashCode, notify, notifyAll, toString, wait, wait, wait

Field Detail

MUESTREO

public static int **MUESTREO**

Tiempo(ms) que duerme el hilo del objeto entre iteraciones consecutivas. Tasa de muestreo con la que se realizan las operaciones de este objeto.

---

representar

public Representar **representar**

Donde se representará la ventana de diálogo.

See Also:

Representar

---

id_dialogo

`public int` **`id_dialogo`**

Identificador del dialogo

---

conversacion

`public` `Conversacion` **`conversacion`**

Elemento conversación que caracteriza al objeto diálogo correspondiente

See Also:

`Conversacion`

---

ind_preg

`public int` **`ind_preg`**

Variable auxiliar que lleva la cuenta del numero de pregunta por el que nos encontramos en una bateria de estas de un test.

---

nota

`public int` **`nota`**

Variable que guarda el numero de fallos que se van teniendo a lo largo de la realización de un test te modo que si se llegase a un limite especificado se optaria por suspender este.

---

maquinaestados

`public` `TransicionEstados` **`maquinaestados`**

Objeto que engloba la maquina de estados principal del juego, y apartir del cual se podrá acceder a métodos para la transición entre estos,...

See Also:

`TransicionEstados`

---

vivo

`public boolean` **`vivo`**

Variable que nos dirá si el hilo está vivo

Constructor Detail

Dialogo

`public` **`Dialogo`**`(int iddial, Conversacion conversacion)`

Constructor de un diálogo caracterizado por un identificador concreto. Dicho dialogo dentra un titulo determinado, un color de fondo concreto (en el caso de ser null la entrada se optará por elegir el color por defecto), una tira de imágenes (a igual que antes en el caso de ser null se optará por elegir la imagen por defecto), y una conversación fija. El constructor creará la pantalla a para configurarla a medida que avanza la conversación. Además cargará las imágenes que formrán la animación una vez se empiece a ejecutar el hilo (run).

Parameters:

`iddial` - Identificador del dialogo

`titulo` - Titulo de la ventana de dialogo

`color_fondo` - Color de fondo de la ventana

`conversacion` - Conversación que regirá la ventana de dialogo

`imagenesg` - Array de imagenes que constituyen la animación una vez se muestran todas de forma consecutiva

See Also:

`run()`

Dialogo

`public` **`Dialogo`**`()`

Constructor por defecto

Method Detail

setRepresentar

`public void` **`setRepresentar`**`(Representar r)`

Metodo con el cual asignamos dónde se va a representar nuestro dialogo

Parameters:

`r` -

See Also:

`Representar`

---

setMaquinaEstados

`public void` **`setMaquinaEstados`**`(TransicionEstados maquinaestados)`

Selecciónа la maquina de estados que caracteriza el Juego.

Parameters:

`maquinaestados` - Maquina de estados que asignamos al Dialogo

See Also:

`TransicionEstados`, `maquinaestados`

---

activo

`public boolean` **`activo`**`()`

Comprueba si el hilo sigue vivo

---

areaTexto

`public void` **`areaTexto`**`()`

Metodo por el que crearemos una ventana de dialogo simple, es decir, en la que no haya interacción con el usuario, simplemente el personaje dirá su texto sin dé ninguna opción al jugador

---

opciones

```
public void opciones(Intervencion interv,
                     int num,
                     boolean test,
                     int indice)
```

Metodo con el que crearemos opciones en la ventana de dialogo

Parameters:

`interv` - Intervencion del dialogo

`num` - numero de opciones de la intervencion

`test,` - true si estamos en un test, false si no es asi

`indice` - indice donde podremos encontrar las opciones de la intervencion

See Also:

`Intervencion`

---

cincoOpciones

```
public void cincoOpciones(Intervencion interv,
                          boolean test,
                          int indice)
```

Metodo por el cual se crea una ventana con cinco opciones

Parameters:

`interv` - Intervencion del dialogo

`test` - indica si estamos en un test o en un texto donde puedes elegir una opcion entre varias

`indice` - indice donde podemos encontrar las opciones que tendra este dialogo

---

dosOpciones

```
public void dosOpciones(Intervencion interv,
                        boolean test,
                        int indice)
```

Metodo por el cual se crea una ventana con dos opciones

Parameters:

`interv` - Intervencion del dialogo

`test` - indica si estamos en un test o en un texto donde puedes elegir una opcion entre varias

`indice` - indice donde podemos encontrar las opciones que tendra este dialogo

---

tresOpciones

```
public void tresOpciones(Intervencion interv,
                         boolean test,
                         int indice)
```

Metodo por el cual se crea una ventana con tres opciones

Parameters:

`interv` - Intervencion del dialogo

`test` - indica si estamos en un test o en un texto donde puedes elegir una opcion entre varias

`indice` - indice donde podemos encontrar las opciones que tendra este dialogo

---

cuatroOpciones

```
public void cuatroOpciones(Intervencion interv,
                           boolean test,
                           int indice)
```

Metodo por el cual se crea una ventana con cuatro opciones

Parameters:

`interv` - Intervencion del dialogo

`test` - indica si estamos en un test o en un texto donde puedes elegir una opcion entre varias

`indice` - indice donde podemos encontrar las opciones que tendra este dialogo

---

setTexto

```
public void setTexto(Etiqueta etiqueta,
                     java.lang.String label)
```

Modifica el texto que contiene la ventana de dialogo

Parameters:

`etiqueta` - Etiqueta que se modificará

`label` - nuevo texto que tendrá dicha etiqueta

---

ocultar

```
public void ocultar()
```

Oculta la ventana en caso de que sea visible

---

setVisible

```
public void setVisible()
```

Metodo que hace que la ventana sea visible en el juego

---

isVisible

```
public boolean isVisible()
```

Comprueba si la ventana es visible

Returns:

si es visible o no

---

evaluateInput

```
public void evaluateInput(MouseMapper mouse,
                          com.threed.jpct.util.KeyMapper keyMapper)
```

Comprueba si ha ocurrido algún evento en el diálogo

---

draw

```
public void draw(com.threed.jpct.FrameBuffer buffer)
```

Dibuja la ventana de diálogo

---

elementChanged

```
public void elementChanged(java.lang.String label,
                           java.lang.String data)
```

Metodo encargado de gestionar los eventos producidos una vez se ha pulsado algun boton. Este metodo gestionará estos eventos reaccionando consecuentemente con cada uno de ellos, es decir, pasando de estados en caso necesario, pasando a la pregunta siguiente en el caso de los test, o a la intervencion siguiente. En caso de pulsar salir, el metodo hará lo necesario para un cierre ordenado de la pantalla.

Specified by:

`elementChanged` in interface `GUIListener`

Parameters:

`evento` - Evento que produjo la ejecución del metodo

`arg` - Descripcion de dicho evento

---

setSonido

```
public void setSonido(boolean sonido)
```

Metodo por el que indicamos si hay que reproducir el sonido o no

Parameters:

`true` - si se reproduce el sonido, false en caso contrario

---

sonido

```
public void sonido()
```

Metodo por el que reproducimos el sonido de la intervencion actual

---

run

```
public void run()
```

Metodo que inicia el diálogo, y mientras vivo sea true refrescará la imagenes de la animación cada cierto tiempo impuesto por MUESTREO, produciendo así una animación. En caso de contrar con una sola imagen, no se refrescará esta.

Specified by:

`run` in interface `java.lang.Runnable`

Class Docencia

```
java.lang.Object
  └ java.awt.Component
      └ java.awt.Container
          └ java.awt.Window
              └ java.awt.Frame
                  └ javax.swing.JFrame
                      └ Docencia
```

All Implemented Interfaces:

java.awt.event.ActionListener, java.awt.image.ImageObserver, java.awt.MenuContainer, java.io.Serializable, java.util.EventListener, javax.accessibility.Accessible, javax.swing.RootPaneContainer, javax.swing.WindowConstants

---

```
public class Docencia
extends javax.swing.JFrame
implements java.awt.event.ActionListener
```

La clase Docencia es la clase con la que podemos introducir los datos docentes de la plataforma del videojuego

Since:

.JDK 1.4

See Also:

`Cuestionario`, `Temario`, Serialized Form

Nested Class Summary

Nested classes/interfaces inherited from class javax.swing.JFrame
`javax.swing.JFrame.AccessibleJFrame`

Nested classes/interfaces inherited from class java.awt.Frame
`java.awt.Frame.AccessibleAWTFrame`

Nested classes/interfaces inherited from class java.awt.Window
`java.awt.Window.AccessibleAWTWindow`

Nested classes/interfaces inherited from class java.awt.Container
`java.awt.Container.AccessibleAWTContainer`

Nested classes/interfaces inherited from class java.awt.Component
java.awt.Component.AccessibleAWTComponent, java.awt.Component.BaselineResizeBehavior, java.awt.Component.BltBufferStrategy, java.awt.Component.FlipBufferStrategy

Field Summary

Fields inherited from class javax.swing.JFrame
accessibleContext, EXIT_ON_CLOSE, rootPane, rootPaneCheckingEnabled

Fields inherited from class java.awt.Frame
CROSSHAIR_CURSOR, DEFAULT_CURSOR, E_RESIZE_CURSOR, HAND_CURSOR, ICONIFIED, MAXIMIZED_BOTH, MAXIMIZED_HORIZ, MAXIMIZED_VERT, MOVE_CURSOR, N_RESIZE_CURSOR, NE_RESIZE_CURSOR, NORMAL, NW_RESIZE_CURSOR, S_RESIZE_CURSOR, SE_RESIZE_CURSOR, SW_RESIZE_CURSOR, TEXT_CURSOR, W_RESIZE_CURSOR, WAIT_CURSOR

Fields inherited from class java.awt.Component
BOTTOM_ALIGNMENT, CENTER_ALIGNMENT, LEFT_ALIGNMENT, RIGHT_ALIGNMENT, TOP_ALIGNMENT

Fields inherited from interface javax.swing.WindowConstants
DISPOSE_ON_CLOSE, DO_NOTHING_ON_CLOSE, HIDE_ON_CLOSE

Fields inherited from interface java.awt.image.ImageObserver
ABORT, ALLBITS, ERROR, FRAMEBITS, HEIGHT, PROPERTIES, SOMEBITS, WIDTH

Constructor Summary
Docencia() Constructor de la clase

Method Summary	
void	**actionPerformed**(java.awt.event.ActionEvent e) Manejador tras pulsar botones
void	**agregarTema**(Tema nuevo) Metodo por el cual agregamos un tema docente
void	**agregarTest**(Test nuevo) Metodo por el cual agregamos un test
void	**borrarTema**(int index) Metodo por el cual eliminamos un tema de la aplicacion
void	**borrarTest**(int index) Metodo por el cual eliminamos un test de la aplicacion
void	**dibujarTablaTema**() Metodo por el cual actualizamos el contenido de la tabla de temas de la aplicacion
void	**dibujarTablaTest**() Metodo por el cual actualizamos el contenido de la tabla de tests de la aplicacion
boolean	**getModifica**() Metodo por el cual sabemos si lo que queremos hacer es modificar un tema
boolean	**getModificaTest**() Metodo por el cual sabemos si lo que queremos hacer es modificar un test
Tema	**getTema**(int index) Metodo por el cual obtenemos el tema que hemos señalado en la tabla
Test	**getTest**(int index) Metodo por el cual obtenemos el test que hemos señalado en la tabla
static void	**main**(java.lang.String[] args) Para poder ejecutar la aplicacion docente
void	**modificarTema**(int index, Tema tem) Metodo por el cual modificamos un tema de la aplicacion
void	**modificarTest**(int index, Test tem) Metodo por el cual modificamos un test de la aplicacion
void	**mouseClicked**(java.awt.event.MouseEvent e)
void	**mouseEntered**(java.awt.event.MouseEvent e)
void	**mouseExited**(java.awt.event.MouseEvent e)
void	**mousePressed**(java.awt.event.MouseEvent e)
void	**mouseReleased**(java.awt.event.MouseEvent e)

Class Enunciado

```
java.lang.Object
  └ java.awt.Component
      └ java.awt.Container
          └ java.awt.Window
              └ java.awt.Dialog
                  └ javax.swing.JDialog
                      └ Enunciado
```

All Implemented Interfaces:

java.awt.event.ActionListener, java.awt.image.ImageObserver, java.awt.MenuContainer, java.io.Serializable, java.util.EventListener, javax.accessibility.Accessible, javax.swing.RootPaneContainer, javax.swing.WindowConstants

---

```
public class Enunciado
extends javax.swing.JDialog
implements java.awt.event.ActionListener
```

La clase Enunciado representa la ventana de la aplicacion donde puedes añadir/modificar una pregunta del test, mostrando las opciones de respuesta y las respuestas correctas

See Also:

Serialized Form

Nested Class Summary

Nested classes/interfaces inherited from class javax.swing.JDialog

`javax.swing.JDialog.AccessibleJDialog`

Nested classes/interfaces inherited from class java.awt.Dialog

`java.awt.Dialog.AccessibleAWTDialog, java.awt.Dialog.ModalExclusionType, java.awt.Dialog.ModalityType`

Nested classes/interfaces inherited from class java.awt.Window

`java.awt.Window.AccessibleAWTWindow`

Nested classes/interfaces inherited from class java.awt.Container

`java.awt.Container.AccessibleAWTContainer`

Nested classes/interfaces inherited from class java.awt.Component

`java.awt.Component.AccessibleAWTComponent, java.awt.Component.BaselineResizeBehavior, java.awt.Component.BltBufferStrategy, java.awt.Component.FlipBufferStrategy`

Field Summary

Fields inherited from class javax.swing.JDialog

`accessibleContext, rootPane, rootPaneCheckingEnabled`

Fields inherited from class java.awt.Dialog

`DEFAULT_MODALITY_TYPE`

Fields inherited from class java.awt.Component

`BOTTOM_ALIGNMENT, CENTER_ALIGNMENT, LEFT_ALIGNMENT, RIGHT_ALIGNMENT, TOP_ALIGNMENT`

Fields inherited from interface javax.swing.WindowConstants

`DISPOSE_ON_CLOSE, DO_NOTHING_ON_CLOSE, EXIT_ON_CLOSE, HIDE_ON_CLOSE`

Fields inherited from interface java.awt.image.ImageObserver

`ABORT, ALLBITS, ERROR, FRAMEBITS, HEIGHT, PROPERTIES, SOMEBITS, WIDTH`

Constructor Summary

Enunciado(NuevoTest n)
Constructor de la clase

Enunciado(NuevoTest n, Pregunta p, int index)
Constructor de la clase

Enunciado(NuevoTest n, Pregunta p, int index, boolean editable)
Constructor de la clase

Method Summary	
void	**actionPerformed**(java.awt.event.ActionEvent e) Manejador de botones
void	**iniciar**(Pregunta preg) Metodo que muestra la interfaz grafica de la pregunta del test

Methods inherited from class javax.swing.JDialog

addImpl, createRootPane, dialogInit, getAccessibleContext, getContentPane, getDefaultCloseOperation, getGlassPane, getGraphics, getJMenuBar, getLayeredPane, getRootPane, getTransferHandler, isDefaultLookAndFeelDecorated, isRootPaneCheckingEnabled, paramString, processWindowEvent, remove, repaint, setContentPane, setDefaultCloseOperation, setDefaultLookAndFeelDecorated, setGlassPane, setJMenuBar, setLayeredPane, setLayout, setRootPane, setRootPaneCheckingEnabled, setTransferHandler, update

Methods inherited from class java.awt.Dialog

addNotify, getModalityType, getTitle, hide, isModal, isResizable, isUndecorated, setModal, setModalityType, setResizable, setTitle, setUndecorated, setVisible, show, toBack

Methods inherited from class java.awt.Window

addPropertyChangeListener, addPropertyChangeListener, addWindowFocusListener, addWindowListener, addWindowStateListener, applyResourceBundle, applyResourceBundle, createBufferStrategy, createBufferStrategy, dispose, getBufferStrategy, getFocusableWindowState, getFocusCycleRootAncestor, getFocusOwner, getFocusTraversalKeys, getGraphicsConfiguration, getIconImages, getInputContext, getListeners, getLocale, getModalExclusionType, getMostRecentFocusOwner, getOwnedWindows, getOwner, getOwnerlessWindows, getToolkit, getWarningString, getWindowFocusListeners, getWindowListeners, getWindows, getWindowStateListeners, isActive, isAlwaysOnTop, isAlwaysOnTopSupported, isFocusableWindow, isFocusCycleRoot, isFocused, isLocationByPlatform, isShowing, pack, postEvent, processEvent, processWindowFocusEvent, processWindowStateEvent, removeNotify, removeWindowFocusListener, removeWindowListener, removeWindowStateListener, reshape, setAlwaysOnTop, setBounds, setBounds, setCursor, setFocusableWindowState, setFocusCycleRoot, setIconImage, setIconImages, setLocationByPlatform, setLocationRelativeTo, setMinimumSize, setModalExclusionType, setSize, setSize, toFront

Methods inherited from class java.awt.Container

add, add, add, add, add, addContainerListener, applyComponentOrientation, areFocusTraversalKeysSet, countComponents, deliverEvent, doLayout, findComponentAt, findComponentAt, getAlignmentX, getAlignmentY, getComponent, getComponentAt, getComponentAt, getComponentCount, getComponents, getComponentZOrder, getContainerListeners, getFocusTraversalPolicy, getInsets, getLayout, getMaximumSize, getMinimumSize, getMousePosition, getPreferredSize, insets, invalidate, isAncestorOf, isFocusCycleRoot, isFocusTraversalPolicyProvider, isFocusTraversalPolicySet, layout, list, list, locate, minimumSize, paint, paintComponents, preferredSize, print, printComponents, processContainerEvent, remove, removeAll, removeContainerListener, setComponentZOrder, setFocusTraversalKeys, setFocusTraversalPolicy, setFocusTraversalPolicyProvider, setFont, transferFocusBackward, transferFocusDownCycle, validate, validateTree

Methods inherited from class java.awt.Component

action, add, addComponentListener, addFocusListener, addHierarchyBoundsListener, addHierarchyListener, addInputMethodListener, addKeyListener, addMouseListener, addMouseMotionListener, addMouseWheelListener, bounds, checkImage, checkImage, coalesceEvents, contains, contains, createImage, createImage, createVolatileImage, createVolatileImage, disable, disableEvents, dispatchEvent, enable, enable, enableEvents, enableInputMethods, firePropertyChange, firePropertyChange, firePropertyChange, firePropertyChange, firePropertyChange, firePropertyChange, firePropertyChange, firePropertyChange, firePropertyChange, getBackground, getBaseline, getBaselineResizeBehavior, getBounds, getBounds, getColorModel, getComponentListeners, getComponentOrientation, getCursor, getDropTarget, getFocusListeners, getFocusTraversalKeysEnabled, getFont, getFontMetrics, getForeground, getHeight, getHierarchyBoundsListeners, getHierarchyListeners, getIgnoreRepaint, getInputMethodListeners, getInputMethodRequests, getKeyListeners, getLocation, getLocation, getLocationOnScreen, getMouseListeners, getMouseMotionListeners, getMousePosition, getMouseWheelListeners, getName, getParent, getPeer, getPropertyChangeListeners, getPropertyChangeListeners, getSize, getSize, getTreeLock, getWidth, getX, getY, gotFocus, handleEvent, hasFocus, imageUpdate, inside, isBackgroundSet, isCursorSet, isDisplayable, isDoubleBuffered, isEnabled, isFocusable, isFocusOwner, isFocusTraversable, isFontSet, isForegroundSet, isLightweight, isMaximumSizeSet, isMinimumSizeSet, isOpaque, isPreferredSizeSet, isValid, isVisible, keyDown, keyUp, list, list, list, location, lostFocus, mouseDown, mouseDrag, mouseEnter, mouseExit, mouseMove, mouseUp, move, nextFocus, paintAll, prepareImage, prepareImage, printAll, processComponentEvent, processFocusEvent, processHierarchyBoundsEvent, processHierarchyEvent, processInputMethodEvent, processKeyEvent, processMouseEvent, processMouseMotionEvent, processMouseWheelEvent, remove, removeComponentListener, removeFocusListener, removeHierarchyBoundsListener, removeHierarchyListener, removeInputMethodListener, removeKeyListener, removeMouseListener, removeMouseMotionListener, removeMouseWheelListener, removePropertyChangeListener, removePropertyChangeListener, repaint, repaint, repaint, requestFocus, requestFocus, requestFocusInWindow, requestFocusInWindow, resize, resize, setBackground, setComponentOrientation, setDropTarget, setEnabled, setFocusable, setFocusTraversalKeysEnabled, setForeground, setIgnoreRepaint, setLocale, setLocation, setLocation, setMaximumSize, setName, setPreferredSize, show, size, toString, transferFocus, transferFocusUpCycle

Methods inherited from class java.lang.Object

clone, equals, finalize, getClass, hashCode, notify, notifyAll, wait, wait, wait

Constructor Detail

Enunciado

`public Enunciado(NuevoTest n)`

Constructor de la clase

Parameters:

`n` - es el objeto NuevoTest que invoca esta clase

Enunciado

```
public Enunciado(NuevoTest n,
                 Pregunta p,
                 int index)
```

Constructor de la clase

Parameters:

`n` - es el objeto NuevoTest que invoca esta clase

`p` - es contiene los datos de la pregunta que se mostrará en la ventana de esta clase

`index` - es la posicion que ocupa dicha pregunta en el test

Enunciado

```
public Enunciado(NuevoTest n,
                 Pregunta p,
                 int index,
                 boolean editable)
```

Constructor de la clase

Parameters:

`n` - es el objeto que invoca esta clase

`p` - es la pregunto que tiene los datos a mostrar

`index` - es la posicion que ocupa dicha pregunta en el test

`editable` - indica si los datos de la pregunta se pueden modificar o no

Method Detail

iniciar

`public void iniciar(Pregunta preg)`

Metodo que muestra la interfaz grafica de la pregunta del test

Parameters:

`preg` - es el objeto que contiene todos los datos a mostrar en la ventana, esto es, el enunciado, las opciones y las respuestas correctas, llamamos a este metodo cuando queremos modificar un enunciado seleccionado

---

actionPerformed

`public void actionPerformed(java.awt.event.ActionEvent e)`

Manejador de botones

Specified by:

`actionPerformed` in interface `java.awt.event.ActionListener`

Class Etiqueta

```
java.lang.Object
  └ GUIComponent
      └ Etiqueta
```

```
public class Etiqueta
extends GUIComponent
```

Una etiqueta en la aplicación. Esta clase sirve para mostrar texto por pantalla

---

Field Summary

Fields inherited from class GUIComponent

`caracter`, `visible`

Constructor Summary

Etiqueta(int xpos, int ypos) Constructor de la etiqueta

Method Summary

void	**bajar**() Metodo utilizado para desplazar el texto en una ventana cuando pulsamos en el boton de bajar para ver lo que habrá posteriormente a lo que se muestra por pantalla
void	**draw**(com.threed.jpct.FrameBuffer buffer) Método por el cual dibujamos la etiq

boolean	**evaluateInput**(MouseMapper mouse, com.threed.jpct.util.KeyMapper keyMapper) Método que comprueba si se ha producido un evento
void	**setTexto**() Modifica el texto de la etiqueta introduciendo en dicho texto saltos de linea para que se vea bien en la aplicación
void	**setTexto**(java.lang.String texto) Modifica el texto de la etiqueta.
void	**setX**(int xp) Modifica la posicion X de la etiqueta
void	**setY**(int yp) Modifica la posicion Y de la etiqueta
void	**subir**() Metodo utilizado para desplazar el texto en una ventana cuando pulsamos en el boton de subir para ver lo que había anteriormente

Methods inherited from class GUIComponent

add, getParent, getParentX, getParentY, getX, getY, isVisible, remove, setCaracter, setVisible, setXFigura, setYFigura

Methods inherited from class java.lang.Object

clone, equals, finalize, getClass, hashCode, notify, notifyAll, toString, wait, wait, wait

Constructor Detail

Etiqueta

```
public Etiqueta(int xpos,
                int ypos)
```

Constructor de la etiqueta

Parameters:

xpos - es la posicion x del inicio de la etiqueta

ypos - es la posicion y del inicio de la etiqueta

Method Detail

setY

```
public void setY(int yp)
```

Modifica la posicion Y de la etiqueta

Overrides:

setY in class GUIComponent

Parameters:

la - nueva coordenada Y

---

setX

```
public void setX(int xp)
```

Modifica la posicion X de la etiqueta

Overrides:

setX in class GUIComponent

Parameters:

la - nueva coordenada X

---

setTexto

```
public void setTexto(java.lang.String texto)
```

Modifica el texto de la etiqueta.

Parameters:

texto - es el nuevo texto que tendrá la etiqueta

---

setTexto

```
public void setTexto()
```

Modifica el texto de la etiqueta introduciendo en dicho texto saltos de linea para que se vea bien en la aplicación

---

subir

```
public void subir()
```

Metodo utilizado para desplazar el texto en una ventana cuando pulsamos en el boton de subir para ver lo que había anteriormente

---

bajar

```
public void bajar()
```

Metodo utilizado para desplazar el texto en una ventana cuando pulsamos en el boton de bajar para ver lo que habrá posteriormente a lo que se muestra por pantalla

---

evaluateInput

```
public boolean evaluateInput(MouseMapper mouse,
                             com.threed.jpct.util.KeyMapper keyMapper)
```

Método que comprueba si se ha producido un evento

Overrides:

evaluateInput in class GUIComponent

Parameters:

mouse - maneja el ratón

`keyMapper` - maneja el teclado
Returns:
boolean true si ocurrio un evento, false en caso contrario

---

draw

`public void draw(com.threed.jpct.FrameBuffer buffer)`

Método por el cual dibujamos la etiq
Overrides:
`draw` in class `GUIComponent`
Parameters:
`buffer` - donde dibujamos

Class Fase

`java.lang.Object`
`└ Fase`

`public class Fase`
`extends java.lang.Object`

Clase que define atributos que caracterizan una fase determinada de la maquina de estados, en la que especifican los componentes que se encuentran activos o no, los dialogos correspondiente a un personaje en dicha fase, y el posicionamiento y rotacion de los personajes y puertas.

Since:
.JDK 1.4
See Also:
`TransicionEstados`, `puertas`, `ascensores`, `personajes`, `dialog_person`, `posicion_person`, `posicion_puerta`, `rotacionY_person`, `rotacionY_puerta`

Field Summary	
`boolean[]`	**`ascensores`** Array booleano correspondiente con todas los ascensores existentes y que tendrá un valor de true en aquellas posiciones que coincidan con aquellos ascensores del array de la clase Juego que estén activas.
`Dialogo[]`	**`dialog_person`** Array de dialogos correspondiente con todos los personajes existentes y que tendrá un valor correspondiente a un determinado dialogo en aquellas posiciones que coincidan con los personajes a los que se les quiera asignar dicho dialogo
`int`	**`id_fase`** Identificador de la fase.
`boolean[]`	**`personajes`** Array booleano correspondiente con todas los personajes existentes y que tendrá un valor de true en aquellas posiciones que coincidan con aquellos personajes del array de la clase Juego que estén activas.
`com.threed.jpct.SimpleVector[]`	**`posicion_person`** Array de Vectores que corresponde a la reubicación de un determinado personaje en una fase determinada
`com.threed.jpct.SimpleVector[]`	**`posicion_puerta`** Array de Vectores que corresponde a la reubicación de un determinado elemento (de la clase Puerta) en una fase determinada
`boolean[]`	**`puertas`** Array booleano correspondiente con todas las puertas existentes y que tendrá un valor de true en aquellas posiciones que coincidan con aquellas puertas del array de la clase Juego que estén activas.
`float[]`	**`rotacionY_person`** Array de Vectores que corresponde a la rotación de un determinado personaje en una fase determinada.
`float[]`	**`rotacionY_puerta`** Array de Vectores que corresponde a la rotacion de un determinado elemento (de la clase puerta) en una fase determinada

Constructor Summary
`Fase``(int id, int numpuertas, int numascen, int numperson)` Constructor de la clase que se encarga de la inicialización de los atributos de la clase con valores inactivos (null, false, o 0)
`Fase``(int id, Puerta[] puertas, Ascensor[] ascensores, Personaje[] personajes, Dialogo[] dialogo)` Constructor de la clase que se encarga de la inicialización de los atributos de la clase con valores inactivos (null, false, o 0)

Method Summary	
`Ascensor[]`	**`getAscensores`**`()` Método por el que obtenemos todos los ascensores que tiene la fase
`Dialogo[]`	**`getDialogos`**`()` Método por el que obtenemos todos los dialogos que tiene la fase
`Personaje[]`	**`getPersonajes`**`()` Método por el que obtenemos todos los personajes que tiene la fase
`Puerta[]`	**`getPuertas`**`()` Método por el que obtenemos todos las puertas que tiene la fase

java.lang.String	**getResumen**() Método por el que obtenemos el resumen de la fase
void	**setAscensores**(Ascensor[] ascen) Método por el que modificamos el array de ascensores que tiene una fase
void	**setPersonajes**(Personaje[] pers) Método por el que modificamos el array de personajes que tiene una fase
void	**setPuertas**(Puerta[] puer) Método por el que modificamos el array de puertas que tiene una fase
void	**setResumen**(java.lang.String resumen) Método por el que modificamos el resumen de la fase

Methods inherited from class java.lang.Object

clone, equals, finalize, getClass, hashCode, notify, notifyAll, toString, wait, wait, wait

Field Detail

id_fase

```
public int id_fase
```

Identificador de la fase. Entero comprendido entre 0 e infinito, nunca con valores negativos El valor -1 es utilizado para referenciar a la fase actual por lo que indicar tal valor collevaria un fallo fatal.

See Also:
TransicionEstados

---

puertas

```
public boolean[] puertas
```

Array booleano correspondiente con todas las puertas existentes y que tendrá un valor de true en aquellas posiciones que coincidan con aquellas puertas del array de la clase Juego que estén activas.

---

ascensores

```
public boolean[] ascensores
```

Array booleano correspondiente con todas los ascensores existentes y que tendrá un valor de true en aquellas posiciones que coincidan con aquellos ascensores del array de la clase Juego que estén activas.

---

personajes

```
public boolean[] personajes
```

Array booleano correspondiente con todas los personajes existentes y que tendrá un valor de true en aquellas posiciones que coincidan con aquellos personajes del array de la clase Juego que estén activas.

---

dialog_person

```
public Dialogo[] dialog_person
```

Array de dialogos correspondiente con todos los personajes existentes y que tendrá un valor correspondiente a un determinado dialogo en aquellas posiciones que coincidan con los personajes a los que se les quiera asignar dicho dialogo

---

posicion_person

```
public com.threed.jpct.SimpleVector[] posicion_person
```

Array de Vectores que corresponde a la reubicación de un determinado personaje en una fase determinada

---

posicion_puerta

```
public com.threed.jpct.SimpleVector[] posicion_puerta
```

Array de Vectores que corresponde a la reubicación de un determinado elemento (de la clase Puerta) en una fase determinada

---

rotacionY_person

```
public float[] rotacionY_person
```

Array de Vectores que corresponde a la rotación de un determinado personaje en una fase determinada.

---

rotacionY_puerta

```
public float[] rotacionY_puerta
```

Array de Vectores que corresponde a la rotacion de un determinado elemento (de la clase puerta) en una fase determinada

Constructor Detail

Fase

```
public Fase(int id,
            int numpuertas,
            int numascen,
            int numperson)
```

Constructor de la clase que se encarga de la inicialización de los atributos de la clase con valores inactivos (null, false, o 0)

Parameters:

`id` - Identificador de la fase

`numpuertas` - Numero de elementos (de la clase Puerta) que han sido creados

`numascen` - Numero de ascensores que han sido creados

`numperson` - Numero de personas que han sido creadas

See Also:
TransicionEstados

Fase

```
public Fase(int id,
            Puerta[] puertas,
            Ascensor[] ascensores,
            Personaje[] personajes,
            Dialogo[] dialogo)
```

Constructor de la clase que se encarga de la inicialización de los atributos de la clase con valores inactivos (null, false, o 0)

Parameters:

`id` - Identificador de la fase

`puertas` - Array de Puertas que han sido creados

`ascensores` - Array de ascensores que han sido creados

`personajes` - Array de personas que han sido creados

`dialogo` - Array de dialogos que han sido creados en la fase

Method Detail

getDialogos

```
public Dialogo[] getDialogos()
```

Método por el que obtenemos todos los dialogos que tiene la fase

Returns:

dialog_person Array de dialogos

---

getPersonajes

```
public Personaje[] getPersonajes()
```

Método por el que obtenemos todos los personajes que tiene la fase

Returns:

pers Array de personajes

---

setPersonajes

```
public void setPersonajes(Personaje[] pers)
```

Método por el que modificamos el array de personajes que tiene una fase

Parameters:

`ascen` - contiene los nuevos personajes de la fase

---

getPuertas

```
public Puerta[] getPuertas()
```

Método por el que obtenemos todos las puertas que tiene la fase

Returns:

puer Array de puertas

---

setPuertas

```
public void setPuertas(Puerta[] puer)
```

Método por el que modificamos el array de puertas que tiene una fase

Parameters:

`puer` - contiene las nuevas puertas de la fase

---

getAscensores

```
public Ascensor[] getAscensores()
```

Método por el que obtenemos todos los ascensores que tiene la fase

Returns:

ascen Array de ascensores

---

setAscensores

```
public void setAscensores(Ascensor[] ascen)
```

Método por el que modificamos el array de ascensores que tiene una fase

Parameters:

`ascen` - contiene los nuevos ascensores de la fase

---

getResumen

```
public java.lang.String getResumen()
```

Método por el que obtenemos el resumen de la fase

Returns:

resumen

---

setResumen

```
public void setResumen(java.lang.String resumen)
```

Método por el que modificamos el resumen de la fase

Parameters:

`resumen` - String que resume la fase

Class Fichero

```
java.lang.Object
  └ Fichero
```

Direct Known Subclasses:

GeneraDocumento, Guardar

```
public class Fichero
extends java.lang.Object
```

Clase con la que obtenemos determinados elementos a traves de pasar como parametro otro elemento para generar un fichero de juego

Constructor Summary
`Fichero()`

Method Summary	
`org.jdom.Element`	`getEscenario(org.jdom.Element raiz)` Metodo que genera un elemento escenario xml segun el dtd de juego, obtiene los valores a traves del elemento raiz que se pasa como parametro
`org.jdom.Element`	`getPantalla(org.jdom.Element raiz)` Metodo que genera un elemento pantalla xml segun el dtd de juego, obtiene los valores a traves del elemento raiz que se pasa como parametro
`org.jdom.Element`	`getProtagonista(org.jdom.Element raiz)` Metodo que genera un elemento protagonista xml segun el dtd de juego, obtiene los valores a traves del elemento raiz que se pasa como parametro
`org.jdom.Element`	`getTemporizador(org.jdom.Element raiz)` Metodo que genera un elemento temporizador xml segun el dtd de juego, obtiene los valores a traves del elemento raiz que se pasa como parametro
`void`	`setCamara(com.threed.jpct.SimpleVector camara)` Método por el que modificamos la posicion del protagonista

Methods inherited from class java.lang.Object
`clone, equals, finalize, getClass, hashCode, notify, notifyAll, toString, wait, wait, wait`

Constructor Detail

Fichero

```
public Fichero()
```

Method Detail

setCamara

```
public void setCamara(com.threed.jpct.SimpleVector camara)
```

Método por el que modificamos la posicion del protagonista

Parameters:

`camara` - SimpleVector que contiene las coordenadas del protagonista

---

getProtagonista

```
public org.jdom.Element getProtagonista(org.jdom.Element raiz)
```

Metodo que genera un elemento protagonista xml segun el dtd de juego, obtiene los valores a traves del elemento raiz que se pasa como parametro

Parameters:

`raiz` - Element del que se obtienen los datos para generar el elemento xml

Returns:

Element Protagonista generado para crear el fichero de juego

---

getTemporizador

```
public org.jdom.Element getTemporizador(org.jdom.Element raiz)
```

Metodo que genera un elemento temporizador xml segun el dtd de juego, obtiene los valores a traves del elemento raiz que se pasa como parametro

Parameters:

`raiz` - Element del que se obtienen los datos para generar el elemento xml

Returns:

Element Temporizador generado para crear el fichero de juego

getPantalla

```
public org.jdom.Element getPantalla(org.jdom.Element raiz)
```

Metodo que genera un elemento pantalla xml segun el dtd de juego, obtiene los valores a traves del elemento raiz que se pasa como parametro

Parameters:

`raiz` - Element del que se obtienen los datos para generar el elemento xml

Returns:

Element Pantalla generado para crear el fichero de juego

---

getEscenario

```
public org.jdom.Element getEscenario(org.jdom.Element raiz)
```

Metodo que genera un elemento escenario xml segun el dtd de juego, obtiene los valores a traves del elemento raiz que se pasa como parametro

Parameters:

`raiz` - Element del que se obtienen los datos para generar el elemento xml

Returns:

Element Escenario generado para crear el fichero de juego

Class GeneraDocumento

```
java.lang.Object
  └ Fichero
      └ GeneraDocumento
```

```
public class GeneraDocumento
extends Fichero
```

Clase con la que generamos el fichero XML que será el guión del juego en caso de estar en el modo edición

Since:
.JDK 1.4

See Also:
`Fichero`

Constructor Summary
`GeneraDocumento``()` Constructor por defecto
`GeneraDocumento``(Fase[] fases)` Constructor de la clase
`GeneraDocumento``(java.util.Vector fas)` Constructor de la clase

Method Summary	
`org.jdom.DocType`	**`crearDocTypeJuego`**`()` Metodo por el cual le decimos al documento xml cual es el dtd que tiene que seguir
`int`	**`cuentaTemas`**`(java.lang.String fichero)` Método con el que contamos los temas que hay en un determinado fichero
`java.lang.String`	**`generaFichero`**`(java.lang.String nombre)` Metodo por el cual generamos el fichero que será el guión del juego

Methods inherited from class Fichero
`getEscenario`, `getPantalla`, `getProtagonista`, `getTemporizador`, `setCamara`

Methods inherited from class java.lang.Object
`clone, equals, finalize, getClass, hashCode, notify, notifyAll, toString, wait, wait, wait`

Constructor Detail

GeneraDocumento

```
public GeneraDocumento(Fase[] fases)
```

Constructor de la clase

Parameters:
`fases` - es el objeto fase que contiene los demas objetos (ascensores, puertas, personajes, dialogos) para poder hacer el guión

---

GeneraDocumento

```
public GeneraDocumento(java.util.Vector fas)
```

Constructor de la clase

Parameters:
`fas` - son las fases con las que obtenemos el resto de objetos

---

GeneraDocumento

```
public GeneraDocumento()
```

Constructor por defecto

Method Detail

crearDocTypeJuego

```
public org.jdom.DocType crearDocTypeJuego()
```

Metodo por el cual le decimos al documento xml cual es el dtd que tiene que seguir

Returns:
doctype del dtd

---

generaFichero

```
public java.lang.String generaFichero(java.lang.String nombre)
```

Metodo por el cual generamos el fichero que será el guión del juego

Parameters:
`nombre` - es el nombre del fichero generado

---

cuentaTemas

```
public int cuentaTemas(java.lang.String fichero)
```

Método con el que contamos los temas que hay en un determinado fichero
Parameters:
`fichero` - es la ruta al fichero donde contamos el numero de temas

Class Guardar

```
java.lang.Object
  └ Fichero
      └ Guardar
```

```
public class Guardar
extends Fichero
```

Clase con la que podemos guardar la partida que estamos jugando

---

Constructor Summary
Guardar(com.threed.jpct.SimpleVector posicion) Constructor de la clase

Method Summary	
org.jdom.DocType	**crearDocType**() Metodo por el cual le decimos al documento xml cual es el dtd que tiene que seguir
void	**guardarPartida**(java.lang.String inicio, java.lang.String nombreFich, int id) Método con el que guardamos una partida.

Methods inherited from class Fichero
getEscenario, getPantalla, getProtagonista, getTemporizador, setCamara

Methods inherited from class java.lang.Object
clone, equals, finalize, getClass, hashCode, notify, notifyAll, toString, wait, wait, wait

Constructor Detail

Guardar

```
public Guardar(com.threed.jpct.SimpleVector posicion)
```

Constructor de la clase
Parameters:
`posicion` - es la posicion del protagonista

Method Detail

crearDocType

```
public org.jdom.DocType crearDocType()
```

Metodo por el cual le decimos al documento xml cual es el dtd que tiene que seguir
Returns:
doctype del dtd

---

guardarPartida

```
public void guardarPartida(java.lang.String inicio,
                           java.lang.String nombreFich,
                           int id)
```

Método con el que guardamos una partida. Genera un fichero donde almacenamos el guion del juego con la partida almacenada
Parameters:
`incio` - es la ruta del guion inicial del juego
`nombreFichero` - es el nombre de salida del guion que creamos
`id` - es el id de la fase en la que nos encontramos

Class Intervencion

```
java.lang.Object
  └ Intervencion
```

```
public class Intervencion
extends java.lang.Object
```

Clase que impleneta atributos y métodos para trabajar, crear y configurar intervenciones que forman un dialogo

Since:

.JDK 1.4

See Also:

Dialogo

Field Summary	
static int	**APROBADO** Posición del hijo que implementa la intervención correspondiente al aprobado de un Test
static int	**COMUN** Común indica que viene una parte en común con otra opción ofrecida
int	**id_proxfase** En el caso de encontrarnos con intervencion-Final, esta variable indica la fase siguiente a la que debe saltar nuestra maquina de estados.
static int	**MODO_CONVERSACION** Modo Conversacion.
static int	**MODO_FINAL** Modo Final.
static int	**MODO_LABEL** Modo label.
static int	**MODO_OPCION** Modo Opcion, donde se da a elegir entre varias opciones para continuar el juego
static int	**MODO_TEMA** Modo Tema, en el que se ofrece un tema a impartir
static int	**MODO_TEST** Modo Test, en el que se ofrecen un numero determado de preguntas de las que se pueden suspender un numero minimo para aprobarlo
int	**nota_suspenso** Variable que guarda el numero de fallos a partir del cual se encuentra suspenso un Test en el caso de tratarse de una intervencion modo test.
int	**num_preg** Variable que guarda el numero de preguntas en el caso de tratarse de una intervencion modo test.
int	**numhijos** Variable que guarda en numero de hijos que tiene la actual intervencio.
java.lang.String[]	**opcion** Variable que guarda las opciones posibles en el caso de tratarse de una intervención modo opcion
java.lang.String[][]	**opciones_test** Variable que guarda el enunciado de las respuestas a todas las preguntas en el caso de tratarse de una intervencion modo test
int	**opcionestest** Variable que guarda el numero de opciones de respuesta en el caso de tratarse de una intervencion modo test.
static int	**PARTE_COMUN** Parte Común es la información que puede ser compartida por varias opciones
java.lang.String[]	**preguntas_test** Variable que guarda el enunciado de las preguntas en el caso de tratarse de una intervencion modo test
boolean[][]	**respuestas_test** Variable que guarda las repuestas a todas las opciones de todas las preguntas en el caso de tratarse de una intervencion modo test.
static int	**SUSPENSO** Posición del hijo que implementa la intervención correspondiente al suspenso de un Test
java.lang.String	**texto** Variable que guarda la cadena de texto que pondremos en la ventana de dialogo
int	**tipo** Tipo de intervencion que regirá el comportamiento de esta.

Constructor Summary
Intervencion(int tipo) Constructor de la clase
Intervencion(int tipo, int numOpc) Constructor de una intervencion (utilizado en el menu objetos para construir una intervencion de tipo opcion)
Intervencion(int nhijos, int tipo, int pregtest, org.jdom.Element raiz_test) Constructor que inicializa los array y variables dependiendo de los valores de entrada que metamos, configurando unicametne las variables necesarias para el modo de operación de la intervención que se está construyendo

Method Summary	
boolean	**addHijo**(Intervencion hij) Metodo que añade el nodo intervencio como hijo del nodo intervencion actual de modo que si llega al numero límite de hijos de una intervención se devolverá un true.
void	**crearTest**() El metodo despliega la lista de preguntas del archivo XML elegido, en caso de que dicho archivo solo contenga una pregunta, repetir reiteradamente dicha preguta, en caso de tener varias preguntas elije una de ellas como inicio y las num_preg siguientes.

boolean	**esFinal**() Devuele si la intervención actual en la conversación es de caracter final de conversación o no lo es.
Intervencion	**getAnterior**() Método con el que obtenemos la intervenición antecesora
java.lang.String	**getArchivo**() Método con el que obtenemos la ruta del fichero de test o tema
int	**getId**() Método con el que devolvemos el id de la intervencion
int	**getIdComun**() Método que devuelve el identificativo de una parte comun
int	**getIdPersonaje**() Método con el que obtenemos el id del personaje al que corresponde la intervención
int	**getIdTema**() Método por el cual obtenemos el identificativo del tema que hay que exponer
int	**getNota**() Método que devuelve el número de fallos permitidos para aprobar
int	**getNumeroPreguntas**() Metodo por el cual obtenemos el numero de preguntas que tiene el test
java.lang.String[]	**getOpciones**() Devuelve un array cuyo contenido es el texto de las opciones
java.lang.String	**getPadre**() Método que devuelve el texto principal de la intervención antecesora
java.lang.String	**getPrincipal**() Devuelve el contenido de una intervencion Conversacion, Label, o el texto inicial de una intervencion Opcion
Intervencion[]	**getRamificacion**() Devuelve el array de hijos de esta intervención
int	**getSiguienteFase**() Método por el cual obtenemos el valor de la siguiente fase a la que podremos acceder
java.lang.String	**getSonido**() Método con el que obtenemos el nombre del fichero que tiene el sonido de la intervención
void	**setAnterior**(Intervencion anterior) Método con el que modificamos la intervencion antecesora
void	**setArchivo**(java.lang.String archivo) Método con el que modificamos la ruta del fichero de test o tema
void	**setId**(int id) Método con el que modificamos el id de la intervención
void	**setIdComun**(int idComun) Método con el que modificamos el identificativo que enlaza con una parte comun
void	**setIdPersonaje**(int idPersonaje) Método con el que modificamos el id del personaje que dice la intervencion
void	**setNota**(int nota) Método que modifica el número de fallos con el que se permite aprobar
void	**setNumeroPreguntas**(int num) Metodo por el cual modificamos el numero de preguntas que tiene el test
void	**setOpciones**(java.util.Vector opciones) Método que almacena en un array el texto de las opciones
void	**setPadre**(java.lang.String padre) Método con el que modificamos el texto principal que representa al antecesor
void	**setPrincipal**(java.lang.String principal) Modifica el contenido de una intervencion Conversacion, Label, o el texto inicial de una intervencion Opcion
void	**setSiguienteFase**(int fase) Método por el cual modificamos el valor de la siguiente fase
void	**setSonido**(java.lang.String sonido) Método con el que modificamos el nombre del fichero que tiene el sonido de la intervencion
void	**setTema**(int tema) Método por el cual modificamos el identificativo del tema a exponer

Methods inherited from class java.lang.Object

clone, equals, finalize, getClass, hashCode, notify, notifyAll, toString, wait, wait, wait

Field Detail

MODO_FINAL

public static final int **MODO_FINAL**

Modo Final. Uno de los nodos finales de una conversación que deriva en un cambio de fase, es decir, de estado en la maquina de estados del juego

See Also:

Constant Field Values

MODO_CONVERSACION

`public static final int` **`MODO_CONVERSACION`**

Modo Conversacion. Información o texto más extenso

See Also:

Constant Field Values

MODO_LABEL

`public static final int` **`MODO_LABEL`**

Modo label. Información o texto reducido

See Also:

Constant Field Values

MODO_OPCION

`public static final int` **`MODO_OPCION`**

Modo Opcion, donde se da a elegir entre varias opciones para continuar el juego

See Also:

Constant Field Values

MODO_TEST

`public static final int` **`MODO_TEST`**

Modo Test, en el que se ofrecen un numero determado de preguntas de las que se pueden suspender un numero minimo para aprobarlo

See Also:

Constant Field Values

APROBADO

`public static final int` **`APROBADO`**

Posición del hijo que implementa la intervención correspondiente al aprobado de un Test

See Also:

Constant Field Values

SUSPENSO

`public static final int` **`SUSPENSO`**

Posición del hijo que implementa la intervención correspondiente al suspenso de un Test

See Also:

Constant Field Values

MODO_TEMA

`public static final int` **`MODO_TEMA`**

Modo Tema, en el que se ofrece un tema a impartir

See Also:

Constant Field Values

COMUN

`public static final int` **`COMUN`**

Común indica que viene una parte en común con otra opción ofrecida

See Also:

Constant Field Values

PARTE_COMUN

`public static final int` **`PARTE_COMUN`**

Parte Común es la información que puede ser compartida por varias opciones

See Also:

Constant Field Values

---

tipo

`public int` **`tipo`**

Tipo de intervencion que regirá el comportamiento de esta.

See Also:

`MODO_FINAL`, `MODO_CONVERSACION`, `MODO_LABEL`, `MODO_OPCION`, `MODO_TEST`, `SUSPENSO`, `APROBADO`, `COMUN`, `PARTE_COMUN`

---

numhijos

`public int` **`numhijos`**

Variable que guarda en numero de hijos que tiene la actual intervencio. Este será de 0 en el caso de tratarse de una intervención final, 1 en el caso de ser label o conversación, 2 en el caso de ser test, y el numero de opciones ofrecidas en el caso de ser una intervencion de la modalidad opcion

---

texto

`public java.lang.String` **`texto`**

Variable que guarda la cadena de texto que pondremos en la ventana de dialogo

---

opcion

`public java.lang.String[]` **`opcion`**

Variable que guarda las opciones posibles en el caso de tratarse de una intervención modo opcion

---

preguntas_test

`public java.lang.String[]` **`preguntas_test`**

Variable que guarda el enunciado de las preguntas en el caso de tratarse de una intervencion modo test

opciones_test

`public java.lang.String[][] opciones_test`

Variable que guarda el enunciado de las respuestas a todas las preguntas en el caso de tratarse de una intervencion modo test

---

respuestas_test

`public boolean[][] respuestas_test`

Variable que guarda las repuestas a todas las opciones de todas las preguntas en el caso de tratarse de una intervencion modo test.

---

nota_suspenso

`public int nota_suspenso`

Variable que guarda el numero de fallos a partir del cual se encuentra suspenso un Test en el caso de tratarse de una intervencion modo test.

---

num_preg

`public int num_preg`

Variable que guarda el numero de preguntas en el caso de tratarse de una intervencion modo test.

---

opcionestest

`public int opcionestest`

Variable que guarda el numero de opciones de respuesta en el caso de tratarse de una intervencion modo test.

---

id_proxfase

`public int id_proxfase`

En el caso de encontrarnos con intervencion-Final, esta variable indica la fase siguiente a la que debe saltar nuestra maquina de estados. Se colocará a -1 en el caso de que la fase a la que queramos saltar sea la misma que en la que nos encontramos ahora.

See Also:

`TransicionEstados`, `Fase.id_fase`

Constructor Detail

Intervencion

```
public Intervencion(int nhijos,
                    int tipo,
                    int pregtest,
                    org.jdom.Element raiz_test)
```

Constructor que inicializa los array y variables dependiendo de los valores de entrada que metamos, configurando unicametne las variables necesarias para el modo de operación de la intervención que se está construyendo

Parameters:

`nhijos` - Numero de hijos que tendrá esta intervencion
`tipo` - Caracter de la intervención actual
`pregtest` - Numero de preguntas de test en el caso de tratarse de una intervencion.test
`opcionestest` - Numero de opciones de respuesta a las preguntas del test en caso de tratarse de una intervención test
`raiz_test` - Elemento raiz que engloba todas las preguntas de test en el archivo XML

See Also:

`MODO_FINAL`, `MODO_CONVERSACION`, `MODO_LABEL`, `MODO_OPCION`, `MODO_TEST`, `Dialogo`, `Conversacion`

Intervencion

`public Intervencion(int tipo)`

Constructor de la clase

Parameters:

`tipo` - es el Tipo de intervención

Intervencion

```
public Intervencion(int tipo,
                    int numOpc)
```

Constructor de una intervencion (utilizado en el menu objetos para construir una intervencion de tipo opcion)

Parameters:

`tipo` - Tipo de intervencion
`numOpc` - numero de opciones de la intervencion (será el número de hijos que tendrá dicha intervención)

Method Detail

setPrincipal

`public void setPrincipal(java.lang.String principal)`

Modifica el contenido de una intervencion Conversacion, Label, o el texto inicial de una intervencion Opcion

Parameters:

`principal` - es el nuevo texto

---

getPrincipal

`public java.lang.String getPrincipal()`

Devuelve el contenido de una intervencion Conversacion, Label, o el texto inicial de una intervencion Opcion

Returns:

texto

---

setOpciones

`public void setOpciones(java.util.Vector opciones)`

Método que almacena en un array el texto de las opciones

Parameters:

`opciones` - contiene el texto a almacenar en un array

---

getOpciones

`public java.lang.String[] getOpciones()`

Devuelve un array cuyo contenido es el texto de las opciones

Returns:

opciones

---

setAnterior

`public void setAnterior(Intervencion anterior)`

Método con el que modificamos la intervencion antecesora

Parameters:

`anterior,` - es la intervención padre

---

getAnterior

`public Intervencion getAnterior()`

Método con el que obtenemos la intervenición antecesora

Returns:

anterior, intervención padre

---

setArchivo

`public void setArchivo(java.lang.String archivo)`

Método con el que modificamos la ruta del fichero de test o tema

Parameters:

`archivo` - nueva ruta

---

getArchivo

`public java.lang.String getArchivo()`

Método con el que obtenemos la ruta del fichero de test o tema

Returns:

archivo, ruta del fichero

---

setTema

`public void setTema(int tema)`

Método por el cual modificamos el identificativo del tema a exponer

Parameters:

`tema` - entero identificativo

---

getIdTema

`public int getIdTema()`

Método por el cual obtenemos el identificativo del tema que hay que exponer

Returns:

tema id del tema

---

setNumeroPreguntas

`public void setNumeroPreguntas(int num)`

Metodo por el cual modificamos el numero de preguntas que tiene el test

Parameters:

`num` - es el numero de preguntas que tiene el test

---

getNumeroPreguntas

`public int getNumeroPreguntas()`

Metodo por el cual obtenemos el numero de preguntas que tiene el test

Parameters:

`numeroPreguntas` -

---

setSiguienteFase

`public void setSiguienteFase(int fase)`

Método por el cual modificamos el valor de la siguiente fase

Parameters:

`fase` - indica la fase a la que iremos

---

getSiguienteFase

`public int getSiguienteFase()`

Método por el cual obtenemos el valor de la siguiente fase a la que podremos acceder

Returns:

siguienteFase

---

setNota

`public void setNota(int nota)`

Método que modifica el número de fallos con el que se permite aprobar

Parameters:

`nota,` - número de fallos permitos para aprobar

---

getNota

`public int getNota()`

Método que devuelve el número de fallos permitidos para aprobar

Returns:

nota, número de fallos permitidos

setIdPersonaje

`public void setIdPersonaje(int idPersonaje)`

Método con el que modificamos el id del personaje que dice la intervencion

Parameters:

`idPersonaje` - es el nuevo id del personaje

---

getIdPersonaje

`public int getIdPersonaje()`

Método con el que obtenemos el id del personaje al que corresponde la intervención

Returns:

idPersonaje

---

setPadre

`public void setPadre(java.lang.String padre)`

Método con el que modificamos el texto principal que representa al antecesor

Parameters:

`padre` - es el nuevo texto principal que identifica al padre de la intervencion

---

getPadre

`public java.lang.String getPadre()`

Método que devuelve el texto principal de la intervención antecesora

Returns:

texto del padre

---

setId

`public void setId(int id)`

Método con el que modificamos el id de la intervención

Parameters:

`id` - es el nuevo id

---

getId

`public int getId()`

Método con el que devolvemos el id de la intervencion

Returns:

id de dicha intervencion

---

setIdComun

`public void setIdComun(int idComun)`

Método con el que modificamos el identificativo que enlaza con una parte comun

Parameters:

`idComun` - nuevo id

---

getIdComun

`public int getIdComun()`

Método que devuelve el identificativo de una parte comun

Returns:

id

---

setSonido

`public void setSonido(java.lang.String sonido)`

Método con el que modificamos el nombre del fichero que tiene el sonido de la intervencion

Parameters:

`sonido` - es el nuevo nombre de fichero

---

getSonido

`public java.lang.String getSonido()`

Método con el que obtenemos el nombre del fichero que tiene el sonido de la intervención

Returns:

sonido, archivo de sonido de la intervencion

---

getRamificacion

`public Intervencion[] getRamificacion()`

Devuelve el array de hijos de esta intervención

Returns:

Devuelve un array de Intervencion que corresponde con sus intervenciones hijas

See Also:

`Conversacion`

---

addHijo

`public boolean addHijo(Intervencion hij)`

Metodo que añade el nodo intervencio como hijo del nodo intervencion actual de modo que si llega al numero límite de hijos de una intervención se devolverá un true. La colocación de dichos hijos es tal que en las intervenciones de opcion se introducen segun el enunciado de las opciones, en el caso de los test se introduce primero el caso del suspenso y segundo el del aprobado.

Parameters:

`hij` - Intervencio que se colgará como hija de la actual

Returns:

Devuelve un booleano que indica si ha llegado ya al numero de hijos a incorporar limite

See Also:

`Conversacion`

esFinal

`public boolean esFinal()`

Devuele si la intervención actual en la conversación es de caracter final de conversación o no lo es.

Returns:
Devuelve true si dicha intervención es final, false en caso contrario.

See Also:
`Conversacion`

---

crearTest

`public void crearTest()`

El metodo despliega la lista de preguntas del archivo XML elegido, en caso de que dicho archivo solo contenga una pregunta, repetir reiteradamente dicha preguta, en caso de tener varias preguntas elije una de ellas como inicio y las num_preg siguientes. Una vez elegidas las preguntas que queremos, completamos los campos que definen la intervención con los valores adecuados (Texto de enunciado,Texto de opciones, valor booleano de las respuestas).

See Also:
`num_preg`, `preguntas_test`, `opciones_test`, `respuestas_test`, `nota_suspenso`

Class Juego

```
java.lang.Object
  └ Juego
```

```
public class Juego
extends java.lang.Object
```

Clase principal encargada de llevar a cabo el analisis e interpretación del archivo XML que construirá el juego. Esta clase se encarga del parseo de dicho archivo así como de la validación de todos los datos contenidos en este, para posteriormente llevar a cabo la implementacion uno por uno de todos los elementos que conforman el juego.

Se definiran datos por defecto imprescindibles en la ejecución del programa, de modo que en el caso de no contar con ellos en el archivo XML, se procederá a configurar el Juego con dichos valores.

Posteriormente a la creación de cada uno de los elementos que conforman el Juego, se procede a ejecutar dicho juego, activando cada uno de los componentes necesarios (representacion, teclado,protagonista,...), para finalmente entrar en un bucle de control de la maquina de estados que estará activo mientras no se pulse la tecla ESC. Dicho bucle de control realizará accesos a la máquina de estados y en caso de ser necesario realizará los cambios de fase correspondiente (activando,desactivando y reubicando los componentes necesarios). Por otro lado dicho control del juego tambien hará uso de los cerrojos disponibles en la clase TransicionEstados de modo que pueda sincronizar la ejecución de partes de codigo que lo requieran.

Since:
.JDK 1.4

See Also:
`TransicionEstados`, `Fase`

Field Summary	
`static int`	**`MUESTREO`** Tiempo(ms) que duerme el hilo del objeto entre iteraciones consecutivas.

Constructor Summary
`Juego``(boolean edic)` Constructor de la clase
`Juego``(java.lang.String nombre_arch_xml, boolean edicion)` Constructor que recibira como parámetro el nombre del archivo xml que sintetiza un juego determinado.

Method Summary	
`static void`	**`main`**`(java.lang.String[] args)` Metodo main.

Methods inherited from class java.lang.Object
`clone, equals, finalize, getClass, hashCode, notify, notifyAll, toString, wait, wait, wait`

Field Detail

MUESTREO

`public static int MUESTREO`

Tiempo(ms) que duerme el hilo del objeto entre iteraciones consecutivas. Tasa de muestreo con la que se realizan las operaciones de este objeto.

Constructor Detail

Juego

```
public Juego(java.lang.String nombre_arch_xml,
             boolean edicion)
```

Constructor que recibira como parámetro el nombre del archivo xml que sintetiza un juego determinado. Este constructor adquirirá el elemento raiz del archivo XML si es posible, mostrando un error en caso contrario. Una vez obtenido, se procederá a construir cada uno de los componentes del juego y llamar al Controlador del Juego por medio del metodo privado Jugar().

Parameters:

`nombre_arch_xml` - nombre del archivo XML que sintetiza el juego

`edicion,` - true si queremos editar el juego, false si queremos jugar

---

Juego

```
public Juego(boolean edic)
```

Constructor de la clase

Parameters:

`edic,` - true si queremos editar el guión del juego, false si queremos jugar

Method Detail

main

```
public static void main(java.lang.String[] args)
```

Metodo main. Metodo que llama al constructor del Juego pasandole como argumento la opcion de si se juega, en caso de omisión se preguntará si editamos o no. Sintaxis: java Juego [-juego || -edicion]

Class MenuObjetos

```
java.lang.Object
  └ Componente
      └ MenuObjetos
```

All Implemented Interfaces:

GUIListener

```
public class MenuObjetos
extends Componente
implements GUIListener
```

Clase con la que se realiza la edición del juego. Gracias a esta clase podemos ir colocando los personajes, puertas y/o ascensores que queremos que tenga el juego, así como el diálogo correspondiente e indicar a través de qué diálogo se abre una fase nueva

Since:

.JDK 1.4

See Also:

`Representar`, `Mundo`, `Teclado`

Field Summary

Fields inherited from class Componente
`activado`, `id_componente`, `objeto`, `posfija`, `rotfija`

Constructor Summary

`MenuObjetos``(Mundo mundo)`	Constructor de la clase

Method Summary

`void`	**`aumentar`**`()` Metodo por el cual aumentamos la escal del objeto 3D que queremos colocar en el escenario
`void`	**`avanzar`**`()` Metodo por el cual movemos un objeto a la derecha del eje Z
`void`	**`bajar`**`()` Metodo por el cual bajamos un objeto (lo movemos en el eje Y)
`void`	**`cargarObjeto`**`(java.lang.String objeto)` Metodo por el cual se cambia el objeto a elegir para colocarlo en el escenario
`void`	**`colocar`**`()` Metodo por el cual colocamos un objeto 3D en el escenario
`void`	**`decrementar`**`()` Metodo por el cual decrementamos la escal del objeto 3D que queremos colocar en el escenario
`void`	**`draw`**`(com.threed.jpct.FrameBuffer buffer)` Metodo que dibuja la ventana de dialogo y todos sus componentes
`void`	**`elementChanged`**`(java.lang.String label, java.lang.String data)` Metodo que maneja los eventos cuando pulsamos sobre un botón de la ventana de diálogo

boolean	**esperar**() Método por el que comprobamos si un personaje debe esperar
void	**evaluateInput**(MouseMapper mouse, com.threed.jpct.util.KeyMapper keyMapper) Metodo por el que sabemos si hemos pinchado con el ratón
boolean	**getAscensor**() Metodo por el cual sabemos si lo que vamos a colocar es un ascensor
boolean	**getCargar**() Metodo que nos indica si tenemos que mostrar la imagen de una figura 3D para elegir
boolean	**getFase**() Metodo que nos dice si no hemos escrito el resumen de una fase
java.lang.String	**getFichero**() Metodo por el que obtenemos el fichero
boolean	**getPersonaje**() Metodo por el cual sabemos si lo que vamos a colocar es un personaje
boolean	**getProbar**() Devuelve la variable que indica si se prueba lo editado
boolean	**getPuerta**() Metodo por el cual sabemos si lo que vamos a colocar es una puerta
Teclado	**getTeclado**() Método por el cual obtenemos el Teclado
void	**girar**() Metodo por el cual giramos un objeto 3D sobre su eje Y
void	**girarX**() Metodo por el cual giramos un objeto 3D sobre su eje X
void	**girarZ**() Metodo por el cual giramos un objeto 3D sobre su eje Z
boolean	**guardar**() Método con el que sabemos si hemos guardado
void	**guardar**(java.lang.String archivo, com.threed.jpct.SimpleVector pos, int idFase) Método con el que guardamos en un fichero el guion del juego editado
boolean	**isVisible**() Metodo que nos dice si la ventana de diálogo es visible o no return true si es visible, false en caso contrario
boolean	**jugar**() Metodo con el que sabemos si hemos terminado la edicion y queremos jugar
void	**moverDerecha**() Metodo por el cual movemos un objeto 3D a la derecha del eje X
void	**moverIzquierda**() Metodo por el cual movemos un objeto 3D a la izquierda del eje X comprueba si el objeto 3D no colisiona con el mundo
void	**perfilarComponente**() Metodo de la clase abstracta
void	**retroceder**() Metodo por el cual movemos un objeto a la izquierda del eje Z
void	**setCamara**(com.threed.jpct.SimpleVector prota) Metodo con el que modificamos la posicion del protagonista
void	**setFases**(java.util.Vector fases) Metodo por el que añadimos fases que se han editado anteriormente para poder seguir con la edición del juego
void	**setFichero**(java.lang.String fichero) Metodo por el cual modificamos el nombre del fichero
void	**setIdPersonaje**(int id) Método por el que modificamos el identificador de un personaje
void	**setInicio**(java.lang.String archivo) Metodo con el que modificamos el archivo inicio
void	**setMundo**(Mundo mundo) Metodo por el cual modificamos el mundo donde añadiremos los objetos 3D
void	**setProbar**(boolean probar) Metodo con el que modificamos la variable que indica si probamos o no
void	**setTeclado**(Teclado tecla) Método por el cual modificamos el teclado
void	**setVisible**(boolean visible) Metodo por el cual podemos hacer visible o no la ventana de diálogo
void	**subir**() Metodo por el cual subimos un objeto (lo movemos en el eje Y)
void	**tipoDialogo**(java.lang.String txt) Metodo que nos da a elegir el tipo de dialogo que tendrá un determinado personaje comprueba antes si el personaje tiene asignado un dialogo

Methods inherited from class Componente

constructorComponente, constructorComponente, getActivacion, getAngulo, getAnguloX, getAnguloZ, getArchivo3D, getDirectorioTexturas, getEscala, getIdComponente, getObjeto, getOrigen, recogerTexturas, reposicionarComponente, rotarX, rotarZ, setAngulo, setAnguloX, setAnguloZ, setArchivo3D, setDirectorioTexturas, setEscala, setObjeto3D, setOrigen

Methods inherited from class java.lang.Object

clone, equals, finalize, getClass, hashCode, notify, notifyAll, toString, wait, wait, wait

Constructor Detail

MenuObjetos

`public MenuObjetos(Mundo mundo)`

Constructor de la clase

Parameters:

`mundo` - donde añadiremos los objetos de la escena

Method Detail

setMundo

`public void setMundo(Mundo mundo)`

Metodo por el cual modificamos el mundo donde añadiremos los objetos 3D

Parameters:

`nuevo` – mundo

---

getFase

`public boolean getFase()`

Metodo que nos dice si no hemos escrito el resumen de una fase

Returns:

true si no la hemos escrito, false si esta escrita

---

setTeclado

`public void setTeclado(Teclado tecla)`

Método por el cual modificamos el teclado

Parameters:

`tecla` - es el nuevo Teclado

---

getTeclado

`public Teclado getTeclado()`

Método por el cual obtenemos el Teclado

Returns:

tecla

---

esperar

`public boolean esperar()`

Método por el que comprobamos si un personaje debe esperar

Returns:

espera

---

setIdPersonaje

`public void setIdPersonaje(int id)`

Método por el que modificamos el identificador de un personaje

Parameters:

`id` - es el nuevo identificador del personaje

---

getCargar

`public boolean getCargar()`

Metodo que nos indica si tenemos que mostrar la imagen de una figura 3D para elegir

Returns:

true si hay que mostrar la imagen, false en caso contrario

---

getPuerta

`public boolean getPuerta()`

Metodo por el cual sabemos si lo que vamos a colocar es una puerta

Returns:

true si es una puerta, false si no lo es

---

getAscensor

`public boolean getAscensor()`

Metodo por el cual sabemos si lo que vamos a colocar es un ascensor

Returns:

true si es un ascensor, false si no lo es

---

getPersonaje

`public boolean getPersonaje()`

Metodo por el cual sabemos si lo que vamos a colocar es un personaje

Returns:

true si es un personaje, false si no lo es

---

perfilarComponente

```
public void perfilarComponente()
```

Metodo de la clase abstracta

Specified by:

`perfilarComponente` in class `Componente`

---

setVisible

```
public void setVisible(boolean visible)
```

Metodo por el cual podemos hacer visible o no la ventana de diálogo

Parameters:

`visible` - indica si la ventana será visible o no (true visible, false invisible)

---

isVisible

```
public boolean isVisible()
```

Metodo que nos dice si la ventana de diálogo es visible o no return true si es visible, false en caso contrario

---

evaluateInput

```
public void evaluateInput(MouseMapper mouse,
                          com.threed.jpct.util.KeyMapper keyMapper)
```

Metodo por el que sabemos si hemos pinchado con el ratón

---

draw

```
public void draw(com.threed.jpct.FrameBuffer buffer)
```

Metodo que dibuja la ventana de dialogo y todos sus componentes

---

aumentar

```
public void aumentar()
```

Metodo por el cual aumentamos la escal del objeto 3D que queremos colocar en el escenario

---

decrementar

```
public void decrementar()
```

Metodo por el cual decrementamos la escal del objeto 3D que queremos colocar en el escenario

---

avanzar

```
public void avanzar()
```

Metodo por el cual movemos un objeto a la derecha del eje Z

---

retroceder

```
public void retroceder()
```

Metodo por el cual movemos un objeto a la izquierda del eje Z

---

subir

```
public void subir()
```

Metodo por el cual subimos un objeto (lo movemos en el eje Y)

---

bajar

```
public void bajar()
```

Metodo por el cual bajamos un objeto (lo movemos en el eje Y)

---

moverIzquierda

```
public void moverIzquierda()
```

Metodo por el cual movemos un objeto 3D a la izquierda del eje X comprueba si el objeto 3D no colisiona con el mundo

---

moverDerecha

```
public void moverDerecha()
```

Metodo por el cual movemos un objeto 3D a la derecha del eje X

---

girar

```
public void girar()
```

Metodo por el cual giramos un objeto 3D sobre su eje Y

---

girarX

```
public void girarX()
```

Metodo por el cual giramos un objeto 3D sobre su eje X

---

girarZ

```
public void girarZ()
```

Metodo por el cual giramos un objeto 3D sobre su eje Z

---

cargarObjeto

```
public void cargarObjeto(java.lang.String objeto)
```

Metodo por el cual se cambia el objeto a elegir para colocarlo en el escenario

Parameters:

`objeto` - el tipo de objeto a mostrar: puerta, ascensor, personaje

---

colocar

```
public void colocar()
```

Metodo por el cual colocamos un objeto 3D en el escenario

---

tipoDialogo

```
public void tipoDialogo(java.lang.String txt)
```

Metodo que nos da a elegir el tipo de dialogo que tendrá un determinado personaje comprueba antes si el personaje tiene asignado un dialogo

Parameters:

`txt` - es un mensaje que puede mostrar en la ventana

---

setFases

```
public void setFases(java.util.Vector fases)
```

Metodo por el que añadimos fases que se han editado anteriormente para poder seguir con la edición del juego

Parameters:

`fases` - son las fases generadas anteriormente

---

guardar

```
public boolean guardar()
```

Método con el que sabemos si hemos guardado

---

guardar

```
public void guardar(java.lang.String archivo,
                    com.threed.jpct.SimpleVector pos,
                    int idFase)
```

Método con el que guardamos en un fichero el guion del juego editado

Parameters:

`archivo` - nombre del archivo

`pos` - posicion del protagonista

`idFase` - fase inicial

---

setInicio

```
public void setInicio(java.lang.String archivo)
```

Metodo con el que modificamos el archivo inicio

---

setProbar

```
public void setProbar(boolean probar)
```

Metodo con el que modificamos la variable que indica si probamos o no

---

getProbar

```
public boolean getProbar()
```

Devuelve la variable que indica si se prueba lo editado

---

setCamara

```
public void setCamara(com.threed.jpct.SimpleVector prota)
```

Metodo con el que modificamos la posicion del protagonista

Parameters:

`posicion` - protagonista

---

getFichero

```
public java.lang.String getFichero()
```

Metodo por el que obtenemos el fichero

---

setFichero

```
public void setFichero(java.lang.String fichero)
```

Metodo por el cual modificamos el nombre del fichero

---

jugar

```
public boolean jugar()
```

Metodo con el que sabemos si hemos terminado la edicion y queremos jugar

Returns:

true si queremos jugar, false en caso contrario

---

elementChanged

```
public void elementChanged(java.lang.String label,
                           java.lang.String data)
```

Metodo que maneja los eventos cuando pulsamos sobre un botón de la ventana de diálogo

Specified by:

`elementChanged` in interface `GUIListener`

Parameters:

`label` - es la etiqueta del elemento en cuestion sobre el que ha ocurrido el evento

`data` - son los datos que este elemento sostiene (null si no tiene ninguno)

Class NuevoTema

```
java.lang.Object
  └ java.awt.Component
      └ java.awt.Container
          └ java.awt.Window
              └ java.awt.Dialog
                  └ javax.swing.JDialog
                      └ NuevoTema
```

All Implemented Interfaces:

java.awt.event.ActionListener, java.awt.image.ImageObserver, java.awt.MenuContainer, java.io.Serializable, java.util.EventListener, javax.accessibility.Accessible, javax.swing.RootPaneContainer, javax.swing.WindowConstants

---

```
public class NuevoTema
extends javax.swing.JDialog
implements java.awt.event.ActionListener
```

Ventana que representa los datos de un tema

Since:

.JDK 1.4

See Also:

Tema, Docencia, Serialized Form

Nested Class Summary

Nested classes/interfaces inherited from class javax.swing.JDialog
javax.swing.JDialog.AccessibleJDialog

Nested classes/interfaces inherited from class java.awt.Dialog
java.awt.Dialog.AccessibleAWTDialog, java.awt.Dialog.ModalExclusionType, java.awt.Dialog.ModalityType

Nested classes/interfaces inherited from class java.awt.Window
java.awt.Window.AccessibleAWTWindow

Nested classes/interfaces inherited from class java.awt.Container
java.awt.Container.AccessibleAWTContainer

Nested classes/interfaces inherited from class java.awt.Component
java.awt.Component.AccessibleAWTComponent, java.awt.Component.BaselineResizeBehavior, java.awt.Component.BltBufferStrategy, java.awt.Component.FlipBufferStrategy

Field Summary

Fields inherited from class javax.swing.JDialog
accessibleContext, rootPane, rootPaneCheckingEnabled

Fields inherited from class java.awt.Dialog
DEFAULT_MODALITY_TYPE

Fields inherited from class java.awt.Component
BOTTOM_ALIGNMENT, CENTER_ALIGNMENT, LEFT_ALIGNMENT, RIGHT_ALIGNMENT, TOP_ALIGNMENT

Fields inherited from interface javax.swing.WindowConstants
DISPOSE_ON_CLOSE, DO_NOTHING_ON_CLOSE, EXIT_ON_CLOSE, HIDE_ON_CLOSE
Fields inherited from interface java.awt.image.ImageObserver
ABORT, ALLBITS, ERROR, FRAMEBITS, HEIGHT, PROPERTIES, SOMEBITS, WIDTH

Constructor Summary

NuevoTema(Docencia f) Constructor de la clase

NuevoTema(Docencia f, Tema tem, int index, boolean editable) Constructor de la clase	

Method Summary	
void	**actionPerformed**(java.awt.event.ActionEvent e) Tratamiento de eventos
boolean	**addTema**() Metodo por el cual añadimos un nuevo tema a la aplicacion docente
void	**borrar**() Metodo por el cual borramos el contenido del tema en la pantalla
void	**iniciar**(Tema tem, boolean editable) Metodo con el que formamos la ventana para poder crear o modificar un tema
boolean	**modificarTema**() Metodo por el cual modificamos un tema dado

Methods inherited from class javax.swing.JDialog

addImpl, createRootPane, dialogInit, getAccessibleContext, getContentPane, getDefaultCloseOperation, getGlassPane, getGraphics, getJMenuBar, getLayeredPane, getRootPane, getTransferHandler, isDefaultLookAndFeelDecorated, isRootPaneCheckingEnabled, paramString, processWindowEvent, remove, repaint, setContentPane, setDefaultCloseOperation, setDefaultLookAndFeelDecorated, setGlassPane, setJMenuBar, setLayeredPane, setLayout, setRootPane, setRootPaneCheckingEnabled, setTransferHandler, update

Methods inherited from class java.awt.Dialog

addNotify, getModalityType, getTitle, hide, isModal, isResizable, isUndecorated, setModal, setModalityType, setResizable, setTitle, setUndecorated, setVisible, show, toBack

Methods inherited from class java.awt.Window

addPropertyChangeListener, addPropertyChangeListener, addWindowFocusListener, addWindowListener, addWindowStateListener, applyResourceBundle, applyResourceBundle, createBufferStrategy, createBufferStrategy, dispose, getBufferStrategy, getFocusableWindowState, getFocusCycleRootAncestor, getFocusOwner, getFocusTraversalKeys, getGraphicsConfiguration, getIconImages, getInputContext, getListeners, getLocale, getModalExclusionType, getMostRecentFocusOwner, getOwnedWindows, getOwner, getOwnerlessWindows, getToolkit, getWarningString, getWindowFocusListeners, getWindowListeners, getWindows, getWindowStateListeners, isActive, isAlwaysOnTop, isAlwaysOnTopSupported, isFocusableWindow, isFocusCycleRoot, isFocused, isLocationByPlatform, isShowing, pack, postEvent, processEvent, processWindowFocusEvent, processWindowStateEvent, removeNotify, removeWindowFocusListener, removeWindowListener, removeWindowStateListener, reshape, setAlwaysOnTop, setBounds, setBounds, setCursor, setFocusableWindowState, setFocusCycleRoot, setIconImage, setIconImages, setLocationByPlatform, setLocationRelativeTo, setMinimumSize, setModalExclusionType, setSize, setSize, toFront

Methods inherited from class java.awt.Container

add, add, add, add, add, addContainerListener, applyComponentOrientation, areFocusTraversalKeysSet, countComponents, deliverEvent, doLayout, findComponentAt, findComponentAt, getAlignmentX, getAlignmentY, getComponent, getComponentAt, getComponentAt, getComponentCount, getComponents, getComponentZOrder, getContainerListeners, getFocusTraversalPolicy, getInsets, getLayout, getMaximumSize, getMinimumSize, getMousePosition, getPreferredSize, insets, invalidate, isAncestorOf, isFocusCycleRoot, isFocusTraversalPolicyProvider, isFocusTraversalPolicySet, layout, list, list, locate, minimumSize, paint, paintComponents, preferredSize, print, printComponents, processContainerEvent, remove, removeAll, removeContainerListener, setComponentZOrder, setFocusTraversalKeys, setFocusTraversalPolicy, setFocusTraversalPolicyProvider, setFont, transferFocusBackward, transferFocusDownCycle, validate, validateTree

Methods inherited from class java.awt.Component

action, add, addComponentListener, addFocusListener, addHierarchyBoundsListener, addHierarchyListener, addInputMethodListener, addKeyListener, addMouseListener, addMouseMotionListener, addMouseWheelListener, bounds, checkImage, checkImage, coalesceEvents, contains, contains, createImage, createImage, createVolatileImage, createVolatileImage, disable, disableEvents, dispatchEvent, enable, enable, enableEvents, enableInputMethods, firePropertyChange, firePropertyChange, firePropertyChange, firePropertyChange, firePropertyChange, firePropertyChange, firePropertyChange, firePropertyChange, firePropertyChange, getBackground, getBaseline, getBaselineResizeBehavior, getBounds, getBounds, getColorModel, getComponentListeners, getComponentOrientation, getCursor, getDropTarget, getFocusListeners, getFocusTraversalKeysEnabled, getFont, getFontMetrics, getForeground, getHeight, getHierarchyBoundsListeners, getHierarchyListeners, getIgnoreRepaint, getInputMethodListeners, getInputMethodRequests, getKeyListeners, getLocation, getLocation, getLocationOnScreen, getMouseListeners, getMouseMotionListeners, getMousePosition, getMouseWheelListeners, getName, getParent, getPeer, getPropertyChangeListeners, getPropertyChangeListeners, getSize, getSize, getTreeLock, getWidth, getX, getY, gotFocus, handleEvent, hasFocus, imageUpdate, inside, isBackgroundSet, isCursorSet, isDisplayable, isDoubleBuffered, isEnabled, isFocusable, isFocusOwner, isFocusTraversable, isFontSet, isForegroundSet, isLightweight, isMaximumSizeSet, isMinimumSizeSet, isOpaque, isPreferredSizeSet, isValid, isVisible, keyDown, keyUp, list, list, list, location, lostFocus, mouseDown, mouseDrag, mouseEnter, mouseExit, mouseMove, mouseUp, move, nextFocus, paintAll, prepareImage, prepareImage, printAll, processComponentEvent, processFocusEvent, processHierarchyBoundsEvent, processHierarchyEvent, processInputMethodEvent, processKeyEvent, processMouseEvent, processMouseMotionEvent, processMouseWheelEvent, remove, removeComponentListener, removeFocusListener, removeHierarchyBoundsListener, removeHierarchyListener, removeInputMethodListener, removeKeyListener, removeMouseListener, removeMouseMotionListener, removeMouseWheelListener, removePropertyChangeListener, removePropertyChangeListener, repaint, repaint, repaint, requestFocus, requestFocus, requestFocusInWindow, requestFocusInWindow, resize, resize, setBackground, setComponentOrientation, setDropTarget, setEnabled, setFocusable, setFocusTraversalKeysEnabled, setForeground, setIgnoreRepaint, setLocale, setLocation, setLocation, setMaximumSize, setName, setPreferredSize, show, size, toString, transferFocus, transferFocusUpCycle

Methods inherited from class java.lang.Object
clone, equals, finalize, getClass, hashCode, notify, notifyAll, wait, wait, wait

Constructor Detail

NuevoTema

```
public NuevoTema(Docencia f)
```

Constructor de la clase

Parameters:

`f` - es la ventana que invoca la clase

NuevoTema

```
public NuevoTema(Docencia f,
                 Tema tem,
                 int index,
                 boolean editable)
```

Constructor de la clase

Parameters:

`f` - es la ventana que invoca la clase

`tem` - es el objeto que contiene los datos

`editable` - indica si se pueden editar los datos o no

Method Detail

iniciar

```
public void iniciar(Tema tem,
                    boolean editable)
```

Metodo con el que formamos la ventana para poder crear o modificar un tema

Parameters:

`tem` - contiene los datos a mostrar en la ventana

`editable` - indica si se pueden modificar los datos o no

---

addTema

```
public boolean addTema()
```

Metodo por el cual añadimos un nuevo tema a la aplicacion docente

Returns:

si se añade o no el tema

---

modificarTema

```
public boolean modificarTema()
```

Metodo por el cual modificamos un tema dado

Returns:

si se modifica o no el tema

---

borrar

```
public void borrar()
```

Metodo por el cual borramos el contenido del tema en la pantalla

---

actionPerformed

```
public void actionPerformed(java.awt.event.ActionEvent e)
```

Tratamiento de eventos

Specified by:

`actionPerformed` in interface `java.awt.event.ActionListener`

Parameters:

`e` - indica el boton pulsado

Class NuevoTest

```
java.lang.Object
  └ java.awt.Component
      └ java.awt.Container
          └ java.awt.Window
              └ java.awt.Dialog
                  └ javax.swing.JDialog
                      └ NuevoTest
```

All Implemented Interfaces:

java.awt.event.ActionListener, java.awt.event.MouseListener, java.awt.image.ImageObserver, java.awt.MenuContainer, java.io.Serializable, java.util.EventListener, javax.accessibility.Accessible, javax.swing.RootPaneContainer, javax.swing.WindowConstants

---

```
public class NuevoTest
extends javax.swing.JDialog
implements java.awt.event.ActionListener, java.awt.event.MouseListener
```

La clase NuevoTest permite crear varios tipos de tests que serán utilizados posteriormente en el videojuego para probar al alumno si ha entendido el temario

See Also:

Serialized Form

Nested Class Summary

Nested classes/interfaces inherited from class javax.swing.JDialog
javax.swing.JDialog.AccessibleJDialog

Nested classes/interfaces inherited from class java.awt.Dialog
java.awt.Dialog.AccessibleAWTDialog, java.awt.Dialog.ModalExclusionType, java.awt.Dialog.ModalityType
Nested classes/interfaces inherited from class java.awt.Window
java.awt.Window.AccessibleAWTWindow

Nested classes/interfaces inherited from class java.awt.Container
java.awt.Container.AccessibleAWTContainer

Nested classes/interfaces inherited from class java.awt.Component
java.awt.Component.AccessibleAWTComponent, java.awt.Component.BaselineResizeBehavior, java.awt.Component.BltBufferStrategy, java.awt.Component.FlipBufferStrategy

Field Summary

Fields inherited from class javax.swing.JDialog
accessibleContext, rootPane, rootPaneCheckingEnabled

Fields inherited from class java.awt.Dialog
DEFAULT_MODALITY_TYPE

Fields inherited from class java.awt.Component
BOTTOM_ALIGNMENT, CENTER_ALIGNMENT, LEFT_ALIGNMENT, RIGHT_ALIGNMENT, TOP_ALIGNMENT

Fields inherited from interface javax.swing.WindowConstants
DISPOSE_ON_CLOSE, DO_NOTHING_ON_CLOSE, EXIT_ON_CLOSE, HIDE_ON_CLOSE

Fields inherited from interface java.awt.image.ImageObserver
ABORT, ALLBITS, ERROR, FRAMEBITS, HEIGHT, PROPERTIES, SOMEBITS, WIDTH

Constructor Summary
NuevoTest(Docencia f) Constructor de la clase
NuevoTest(Docencia f, Test tem, int index, boolean editable) Constructor de la clase

Method Summary	
void	**actionPerformed**(java.awt.event.ActionEvent e) Manejador de eventos
void	**addPregunta**(Pregunta p) Metodo por el cual añadimos una pregunta nueva al test
void	**borrar**(int index) Metodo por el cual borramos una pregunta determinada
void	**dibujarTabla**() Metodo por el cual actualizamos los datos de la tabla, cada vez que agregamos una pregunta nueva o modificamos una que ya estaba antes
Pregunta	**getPregunta**(int index) Metodo por el cual obtenemos la pregunta seleccionada en la tabla
void	**iniciar**() Metodo que crea la ventana que nos muestra
void	**modificar**(int index, Pregunta p) Metodo por el cual modificamos una pregunta del test
void	**mouseClicked**(java.awt.event.MouseEvent e)

void	**mouseEntered**(java.awt.event.MouseEvent e)
void	**mouseExited**(java.awt.event.MouseEvent e)
void	**mousePressed**(java.awt.event.MouseEvent e)
void	**mouseReleased**(java.awt.event.MouseEvent e)

Methods inherited from class javax.swing.JDialog

addImpl, createRootPane, dialogInit, getAccessibleContext, getContentPane, getDefaultCloseOperation, getGlassPane, getGraphics, getJMenuBar, getLayeredPane, getRootPane, getTransferHandler, isDefaultLookAndFeelDecorated, isRootPaneCheckingEnabled, paramString, processWindowEvent, remove, repaint, setContentPane, setDefaultCloseOperation, setDefaultLookAndFeelDecorated, setGlassPane, setJMenuBar, setLayeredPane, setLayout, setRootPane, setRootPaneCheckingEnabled, setTransferHandler, update

Methods inherited from class java.awt.Dialog

addNotify, getModalityType, getTitle, hide, isModal, isResizable, isUndecorated, setModal, setModalityType, setResizable, setTitle, setUndecorated, setVisible, show, toBack

Methods inherited from class java.awt.Window

addPropertyChangeListener, addPropertyChangeListener, addWindowFocusListener, addWindowListener, addWindowStateListener, applyResourceBundle, applyResourceBundle, createBufferStrategy, createBufferStrategy, dispose, getBufferStrategy, getFocusableWindowState, getFocusCycleRootAncestor, getFocusOwner, getFocusTraversalKeys, getGraphicsConfiguration, getIconImages, getInputContext, getListeners, getLocale, getModalExclusionType, getMostRecentFocusOwner, getOwnedWindows, getOwner, getOwnerlessWindows, getToolkit, getWarningString, getWindowFocusListeners, getWindowListeners, getWindows, getWindowStateListeners, isActive, isAlwaysOnTop, isAlwaysOnTopSupported, isFocusableWindow, isFocusCycleRoot, isFocused, isLocationByPlatform, isShowing, pack, postEvent, processEvent, processWindowFocusEvent, processWindowStateEvent, removeNotify, removeWindowFocusListener, removeWindowListener, removeWindowStateListener, reshape, setAlwaysOnTop, setBounds, setBounds, setCursor, setFocusableWindowState, setFocusCycleRoot, setIconImage, setIconImages, setLocationByPlatform, setLocationRelativeTo, setMinimumSize, setModalExclusionType, setSize, setSize, toFront

Methods inherited from class java.awt.Container

add, add, add, add, add, addContainerListener, applyComponentOrientation, areFocusTraversalKeysSet, countComponents, deliverEvent, doLayout, findComponentAt, findComponentAt, getAlignmentX, getAlignmentY, getComponent, getComponentAt, getComponentAt, getComponentCount, getComponents, getComponentZOrder, getContainerListeners, getFocusTraversalPolicy, getInsets, getLayout, getMaximumSize, getMinimumSize, getMousePosition, getPreferredSize, insets, invalidate, isAncestorOf, isFocusCycleRoot, isFocusTraversalPolicyProvider, isFocusTraversalPolicySet, layout, list, list, locate, minimumSize, paint, paintComponents, preferredSize, print, printComponents, processContainerEvent, remove, removeAll, removeContainerListener, setComponentZOrder, setFocusTraversalKeys, setFocusTraversalPolicy, setFocusTraversalPolicyProvider, setFont, transferFocusBackward, transferFocusDownCycle, validate, validateTree

Methods inherited from class java.awt.Component

action, add, addComponentListener, addFocusListener, addHierarchyBoundsListener, addHierarchyListener, addInputMethodListener, addKeyListener, addMouseListener, addMouseMotionListener, addMouseWheelListener, bounds, checkImage, checkImage, coalesceEvents, contains, contains, createImage, createImage, createVolatileImage, createVolatileImage, disable, disableEvents, dispatchEvent, enable, enable, enableEvents, enableInputMethods, firePropertyChange, firePropertyChange, firePropertyChange, firePropertyChange, firePropertyChange, firePropertyChange, firePropertyChange, firePropertyChange, getBackground, getBaseline, getBaselineResizeBehavior, getBounds, getBounds, getColorModel, getComponentListeners, getComponentOrientation, getCursor, getDropTarget, getFocusListeners, getFocusTraversalKeysEnabled, getFont, getFontMetrics, getForeground, getHeight, getHierarchyBoundsListeners, getHierarchyListeners, getIgnoreRepaint, getInputMethodListeners, getInputMethodRequests, getKeyListeners, getLocation, getLocation, getLocationOnScreen, getMouseListeners, getMouseMotionListeners, getMousePosition, getMouseWheelListeners, getName, getParent, getPeer, getPropertyChangeListeners, getPropertyChangeListeners, getSize, getSize, getTreeLock, getWidth, getX, getY, gotFocus, handleEvent, hasFocus, imageUpdate, inside, isBackgroundSet, isCursorSet, isDisplayable, isDoubleBuffered, isEnabled, isFocusable, isFocusOwner, isFocusTraversable, isFontSet, isForegroundSet, isLightweight, isMaximumSizeSet, isMinimumSizeSet, isOpaque, isPreferredSizeSet, isValid, isVisible, keyDown, keyUp, list, list, list, location, lostFocus, mouseDown, mouseDrag, mouseEnter, mouseExit, mouseMove, mouseUp, move, nextFocus, paintAll, prepareImage, prepareImage, printAll, processComponentEvent, processFocusEvent, processHierarchyBoundsEvent, processHierarchyEvent, processInputMethodEvent, processKeyEvent, processMouseEvent, processMouseMotionEvent, processMouseWheelEvent, remove, removeComponentListener, removeFocusListener, removeHierarchyBoundsListener, removeHierarchyListener, removeInputMethodListener, removeKeyListener, removeMouseListener, removeMouseMotionListener, removeMouseWheelListener, removePropertyChangeListener, removePropertyChangeListener, repaint, repaint, repaint, requestFocus, requestFocus, requestFocusInWindow, requestFocusInWindow, resize, resize, setBackground, setComponentOrientation, setDropTarget, setEnabled, setFocusable, setFocusTraversalKeysEnabled, setForeground, setIgnoreRepaint, setLocale, setLocation, setLocation, setMaximumSize, setName, setPreferredSize, show, size, toString, transferFocus, transferFocusUpCycle

Methods inherited from class java.lang.Object

clone, equals, finalize, getClass, hashCode, notify, notifyAll, wait, wait, wait

Constructor Detail

NuevoTest

```
public NuevoTest(Docencia f)
```

Constructor de la clase

Parameters:

`f` - es la aplicacion padre que invoca dicha clase

NuevoTest

```
public NuevoTest(Docencia f,
                 Test tem,
                 int index,
                 boolean editable)
```

Constructor de la clase

Parameters:

`f` - es la aplicacion padre que invoca dicha clase

`tem` - es el test que hemos seleccionado con antelacion

`index` - es el indice que representa el test seleccionado

`editable` - true si permitimos modificar los datos, false si no lo permitimos

Method Detail

iniciar

```
public void iniciar()
```

Metodo que crea la ventana que nos muestra

---

addPregunta

```
public void addPregunta(Pregunta p)
```

Metodo por el cual añadimos una pregunta nueva al test

Parameters:

`p` - es la pregunta a añadir

---

dibujarTabla

```
public void dibujarTabla()
```

Metodo por el cual actualizamos los datos de la tabla, cada vez que agregamos una pregunta nueva o modificamos una que ya estaba antes

---

borrar

```
public void borrar(int index)
```

Metodo por el cual borramos una pregunta determinada

Parameters:

`index` - es la fila seleccionada que representa a la pregunta que queremos borrar

---

modificar

```
public void modificar(int index,
                      Pregunta p)
```

Metodo por el cual modificamos una pregunta del test

Parameters:

`index` - es el indice que indica la pregunta que queremos modificar

`p` - es el objeto pregunta que contiene todos los datos de nueva pregunta que sustituira a la que queremos modificar

---

getPregunta

```
public Pregunta getPregunta(int index)
```

Metodo por el cual obtenemos la pregunta seleccionada en la tabla

Parameters:

`index` - es el indice de la pregunta que queremos ver

Returns:

la pregunta deseada

---

actionPerformed

```
public void actionPerformed(java.awt.event.ActionEvent e)
```

Manejador de eventos

Specified by:

`actionPerformed` in interface `java.awt.event.ActionListener`

Parameters:

`e,` - boton presionado

---

mouseClicked

```
public void mouseClicked(java.awt.event.MouseEvent e)
```

Specified by:

`mouseClicked` in interface `java.awt.event.MouseListener`

---

mouseEntered

```
public void mouseEntered(java.awt.event.MouseEvent e)
```

Specified by:

`mouseEntered` in interface `java.awt.event.MouseListener`

---

mouseExited

```
public void mouseExited(java.awt.event.MouseEvent e)
```

Specified by:

mouseExited in interface java.awt.event.MouseListener

---

mousePressed

```
public void mousePressed(java.awt.event.MouseEvent e)
```

Specified by:
mousePressed in interface java.awt.event.MouseListener

---

mouseReleased

```
public void mouseReleased(java.awt.event.MouseEvent e)
```

Specified by:
mouseReleased in interface java.awt.event.MouseListener

Class Personaje

```
java.lang.Object
  └ Componente
      └ Personaje
```

All Implemented Interfaces:
InterfazComponente, java.lang.Runnable

```
public class Personaje
extends Componente
implements InterfazComponente, java.lang.Runnable
```

Clase que implementa atributos y los metodos necesarios para trabajar con el objeto personajes. Dicho objeto tendrá la capacidad de crear dialogos gestionados desde un nuevo hilo cada vez que se ejecuten. El objeto personaje a su vez se ejecutará como un hijo independiente, cuyo funcinamiento consistirá en detectar si se ha producido una colisión en un determinado momento para crear y lanzar la ventana de dialogo en otro nuevo hilo, o bien esperar a que termine un tiempo especificado por MUESTREO para realizar otra nueva comprobación. Esto se repetira mientras el juego este activo y dicho personaje tambien, en caso contrario se suspenderá la ejecución del hilo que fundamenta su funcionamiento.

Since:
.JDK 1.4

See Also:
Componente, InterfazComponente, Dialogo

Field Summary	
boolean	**activado_movimiento** Atributo que indica si se encuentra activado o no el movimiento del personaje En un principio se encuentra activado
static int	**MUESTREO** Tiempo(ms) que duerme el hilo del objeto entre iteraciones consecutivas.
float	**velocidadrotacion** Constante por defecto que indica la velocidad de movimiento del personaje
boolean	**vivo** Variable encargada de conservar vivo el hilo siempre y cuando se encuentre a true.

Fields inherited from class Componente
activado, id_componente, objeto, posfija, rotfija

Constructor Summary
Personaje(int idper, java.lang.String p3ds, java.lang.String ptexturas, float escala) Constructor que recibe por parámetros el id, el directorio del archivo 3ds que conforma el personaje, el directorio de las texturas correspondientes y la escala de dicho objeto, creando así el personaje correspondiente.
Personaje(int idper, java.lang.String p3ds, java.lang.String ptexturas, float escala, java.lang.String eje) Constructor que recibe por parámetros el id, el directorio del archivo 3ds que conforma el personaje, el directorio de las texturas correspondientes, la escala de dicho objeto y el eje por el que caminará creando así el personaje correspondiente.

Method Summary	
void	**activarComponente**() Activa el personaje, reanudando la ejecucion del hilo y haciendo visible el objeto.
void	**desactivarColision**()
void	**desactivarComponente**() Desactiva el ascensor, interrumpiendoo la ejecucion del hilo y haciendo invisible el objeto.
boolean	**getEdicion**() Método con el que obtenemos si estamos editando o no
java.lang.String	**getEje**() Método por el que obtenemos el eje por el que caminará el personaje

float	**getPosicionFinal**() Metodo que devuelve la posicion final
void	**perfilarComponente**() Implementación del método abstracto de la superclase, que termina de definir el objeto 3D, desactivando dicho componente y habilitando las posibles transformaciones posteriores que se puedan realizar respecto a sus ubicacion y rotacion
void	**posicionarComponente**(com.threed.jpct.SimpleVector posicion_ini, float rotacion_ini) Posiciona el elemento 3D en la posicion indicada por el vecotr posicion_ini rotando dicho objeto un valor de ratacion_ini radianes
void	**run**() Metodo que se ejecuta cuando se invoca el metodo start de Thread.
void	**setDialogo**(Dialogo dial) Metodo que da valor al atributo dialogo del objeto Personaje el cual hará referencia al elemento dialogo que caracterizará el comportamiento de un personaje concreto.
void	**setEdicion**(boolean edicion) Método con el que modificamos la variable edicion
void	**setEje**(java.lang.String eje) Método por el que podemos modificar el eje donde se moverá el personaje
void	**setMenuObjetos**(MenuObjetos m) Método con el que modificamos el el objeto MenuObjetos
void	**setPosicionFinal**(float posicion) Método con el que modificamos la posicion final en el eje y
void	**setRecorridoX**(float fin) Método con el que modificamos la posicion final en el eje x
void	**setRecorridoZ**(float fin) Método con el que modificamos la posicion final en el eje z
void	**setRepresentar**(Representar r) Método con el que modificamos el objeto representar
void	**setThread**(java.lang.Thread tt) Metodo que asigna un hilo de ejecución al objeto correspondiente, de modo que se pueda interrumpir, reanudar, finalizar, etc...

Methods inherited from class Componente

constructorComponente, constructorComponente, getActivacion, getAngulo, getAnguloX, getAnguloZ, getArchivo3D, getDirectorioTexturas, getEscala, getIdComponente, getObjeto, getOrigen, recogerTexturas, reposicionarComponente, rotarX, rotarZ, setAngulo, setAnguloX, setAnguloZ, setArchivo3D, setDirectorioTexturas, setEscala, setObjeto3D, setOrigen

Methods inherited from class java.lang.Object

clone, equals, finalize, getClass, hashCode, notify, notifyAll, toString, wait, wait, wait

Field Detail

velocidadrotacion

```
public float velocidadrotacion
```

Constante por defecto que indica la velocidad de movimiento del personaje

See Also:

activado_movimiento, moverPersonaje()

---

vivo

```
public boolean vivo
```

Variable encargada de conservar vivo el hilo siempre y cuando se encuentre a true.

See Also:

run()

MUESTREO

```
public static int MUESTREO
```

Tiempo(ms) que duerme el hilo del objeto entre iteraciones consecutivas. Tasa de muestreo con la que se realizan las operaciones de este objeto.

---

activado_movimiento

```
public boolean activado_movimiento
```

Atributo que indica si se encuentra activado o no el movimiento del personaje En un principio se encuentra activado

See Also:

velocidadrotacion, moverPersonaje()

Constructor Detail

Personaje

```
public Personaje(int idper,
                 java.lang.String p3ds,
                 java.lang.String ptexturas,
                 float escala)
```

Constructor que recibe por parámetros el id, el directorio del archivo 3ds que conforma el personaje, el directorio de las texturas correspondientes y la escala de dicho objeto, creando así el personaje correspondiente.

Parameters:

`idper` - Entero mayor o igual a 0 que identifica al personaje.

p3ds - Archivo 3ds de la figura correspondiente al personaje.
ptexturas - Directorio donde se encuentran ubicadas las texturas correspondientes al 3ds.
escala - Float que indica el tamaño del compoente que se va a insertar.

Personaje

```
public Personaje(int idper,
                 java.lang.String p3ds,
                 java.lang.String ptexturas,
                 float escala,
                 java.lang.String eje)
```

Constructor que recibe por parámetros el id, el directorio del archivo 3ds que conforma el personaje, el directorio de las texturas correspondientes, la escala de dicho objeto y el eje por el que caminará creando así el personaje correspondiente.
Parameters:
idper - Entero mayor o igual a 0 que identifica al personaje.
p3ds - Archivo 3ds de la figura correspondiente al personaje.
ptexturas - Directorio donde se encuentran ubicadas las texturas correspondientes al 3ds.
escala - Float que indica el tamaño del compoente que se va a insertar.
eje - String que indica el eje por el que caminará el personaje.

Method Detail

setEje

```
public void setEje(java.lang.String eje)
```

Método por el que podemos modificar el eje donde se moverá el personaje
Parameters:
eje - String del nuevo eje donde caminará

---

getEje

```
public java.lang.String getEje()
```

Método por el que obtenemos el eje por el que caminará el personaje
Returns:
eje donde camina

---

setRecorridoX

```
public void setRecorridoX(float fin)
```

Método con el que modificamos la posicion final en el eje x
Parameters:
fin - es la posicion final del eje x

---

setRecorridoZ

```
public void setRecorridoZ(float fin)
```

Método con el que modificamos la posicion final en el eje z
Parameters:
fin - es la posicion final del eje z

---

setPosicionFinal

```
public void setPosicionFinal(float posicion)
```

Método con el que modificamos la posicion final en el eje y
Parameters:
fin - es la posicion final del eje x

---

getPosicionFinal

```
public float getPosicionFinal()
```

Metodo que devuelve la posicion final
Returns:
posicionFinal

---

setDialogo

```
public void setDialogo(Dialogo dial)
```

Metodo que da valor al atributo dialogo del objeto Personaje el cual hará referencia al elemento dialogo que caracterizará el comportamiento de un personaje concreto.
Parameters:
dial - Elemento dialogo que queremos instalar en nuestro personaje

---

perfilarComponente

```
public void perfilarComponente()
```

Implementación del método abstracto de la superclase, que termina de definir el objeto 3D, desactivando dicho componente y habilitando las posibles transformaciones posteriores que se puedan realizar respecto a sus ubicacion y rotacion
Specified by:
perfilarComponente in class Componente
See Also:
Componente

---

setThread

```
public void setThread(java.lang.Thread tt)
```

Metodo que asigna un hilo de ejecución al objeto correspondiente, de modo que se pueda interrumpir, reanudar, finalizar, etc...
Parameters:
tt - Thread que se asigna al objeto.

---

posicionarComponente

```
public void posicionarComponente(com.threed.jpct.SimpleVector posicion_ini,
                                 float rotacion_ini)
```

Posiciona el elemento 3D en la posicion indicada por el vecotr posicion_ini rotando dicho objeto un valor de ratacion_ini radianes

Specified by:
`posicionarComponente` in interface `InterfazComponente`

Parameters:
`posicion_ini` - Vector que indica la posicion inicial del objeto.
`rotacion_ini` - float que indica la rotacion inicial del objeto, expresada en radianes.

See Also:
`Componente.reposicionarComponente(com.threed.jpct.SimpleVector, float)`

---

activarComponente

```
public void activarComponente()
```

Activa el personaje, reanudando la ejecucion del hilo y haciendo visible el objeto. Además le hace sensible a colisiones.

Specified by:
`activarComponente` in interface `InterfazComponente`

See Also:
`desactivarComponente()`

---

desactivarComponente

```
public void desactivarComponente()
```

Desactiva el ascensor, interrumpiendoo la ejecucion del hilo y haciendo invisible el objeto. Además le hace insensible a colisiones.

Specified by:
`desactivarComponente` in interface `InterfazComponente`

See Also:
`desactivarComponente()`

---

desactivarColision

```
public void desactivarColision()
```

---

setMenuObjetos

```
public void setMenuObjetos(MenuObjetos m)
```

Método con el que modificamos el el objeto MenuObjetos

Parameters:
`m` - nuevo objeto MenuObjetos

---

setEdicion

```
public void setEdicion(boolean edicion)
```

Método con el que modificamos la variable edicion

Parameters:
`edicion` - es el nuevo valor

---

getEdicion

```
public boolean getEdicion()
```

Método con el que obtenemos si estamos editando o no

Returns:
true si editamos, false en caso contrario

---

setRepresentar

```
public void setRepresentar(Representar r)
```

Método con el que modificamos el objeto representar

Parameters:
`nuevo` - objeto representar

---

run

```
public void run()
```

Metodo que se ejecuta cuando se invoca el metodo start de Thread. Bucle que depende de la variable vivo, y que se encuentra activo mientras esta no sea false. En dichas iteraciones se comprueba si se ha producido alguna colisión, y en tal caso comprueba si el objeto personaje está caracterizado por algun dialogo y de estarlo lo inicia bajo la ejecución de un nuevo hilo quedandose esperando a que termine este. Una vez terminado, se retorna a la ejecución normal del programa abriendo los cerrojos que impedian la ejecución del mismo mientras estaba activo el diálogo.

Antes de esto, y en el caso de que esté activado el movimiento del personaje, realiza una pequeña oscilacion Una vez realizadas o no estas operaciones se esperará un tiempo dado por la variable MUESTREO para volver al inicio del bucle.

Specified by:
`run` in interface `java.lang.Runnable`

See Also:
`vivo`, `MUESTREO`

Class Pregunta

```
java.lang.Object
  └ Pregunta
```

```
public class Pregunta
```

extends java.lang.Object

La clase Pregunta es la clase que representa un enunciado con respuestas, tanto las opciones que nos da, como las respuestas correctas

---

Constructor Summary
Pregunta(java.lang.String enunciado) Constructor de la clase
Pregunta(java.lang.String enunciado, java.lang.String tipo) Constructor de la clase

Method Summary	
void	**addOpcion**(java.lang.String opcion) Metodo por el cual añadimos las opciones de respuestas que tiene una pregunta
void	**addRespuesta**(java.lang.String opcion) Metodo por el cual podemos ver cuales son las respuestas correctas
java.lang.String	**getEnunciado**() Metodo por el cual obtenemos el enunciado de una pregunta determinada
java.util.Vector	**getOpciones**() Metodo por el cual obtenemos las opciones de una pregunta determinada
java.util.Vector	**getRespuestas**() Metodo por el cual obtenemos las respuestas de una pregunta determinada
java.lang.String	**getTipoPregunta**() Metodo por el cual obtenemos el tipo de pregunta que es
void	**setEnunciado**(java.lang.String enunciado) Metodo por el cual podemos modificar el enunciado de una pregunta determinada
void	**setOpciones**(java.util.Vector opciones) Metodo por el cual podemos modificar las opciones de una pregunta determinada
void	**setRespuestas**(java.util.Vector respuestas) Metodo por el cual modificamos las respuestas correctas de una pregunta
void	**setTipoRespuesta**(java.lang.String tipo) Metodo por el cual modificamos el tipo de respuesta del test

Methods inherited from class java.lang.Object
clone, equals, finalize, getClass, hashCode, notify, notifyAll, toString, wait, wait, wait

Constructor Detail

Pregunta

public **Pregunta**(java.lang.String enunciado,
java.lang.String tipo)

Constructor de la clase

Parameters:

enunciado - es el enunciado de la pregunta

Pregunta

public **Pregunta**(java.lang.String enunciado)

Constructor de la clase

Parameters:

enunciado - es el enunciado de la pregunta

Method Detail

addOpcion

public void **addOpcion**(java.lang.String opcion)

Metodo por el cual añadimos las opciones de respuestas que tiene una pregunta

---

addRespuesta

public void **addRespuesta**(java.lang.String opcion)

Metodo por el cual podemos ver cuales son las respuestas correctas

---

setEnunciado

public void **setEnunciado**(java.lang.String enunciado)

Metodo por el cual podemos modificar el enunciado de una pregunta determinada

---

setOpciones

public void **setOpciones**(java.util.Vector opciones)

Metodo por el cual podemos modificar las opciones de una pregunta determinada

Parameters:

opciones - son las nuevas opciones

---

setRespuestas

`public void setRespuestas(java.util.Vector respuestas)`

Metodo por el cual modificamos las respuestas correctas de una pregunta

Parameters:

`respuestas` - son las nuevas respuestas correctas de la pregunta

---

getRespuestas

`public java.util.Vector getRespuestas()`

Metodo por el cual obtenemos las respuestas de una pregunta determinada

Returns:

las respuestas de dicho enunciado

---

getOpciones

`public java.util.Vector getOpciones()`

Metodo por el cual obtenemos las opciones de una pregunta determinada

Returns:

las opciones que nos da el enunciado

---

getEnunciado

`public java.lang.String getEnunciado()`

Metodo por el cual obtenemos el enunciado de una pregunta determinada

Returns:

dicho enunciado

---

getTipoPregunta

`public java.lang.String getTipoPregunta()`

Metodo por el cual obtenemos el tipo de pregunta que es

Returns:

tipo (respuesta libre o con opciones)

---

setTipoRespuesta

`public void setTipoRespuesta(java.lang.String tipo)`

Metodo por el cual modificamos el tipo de respuesta del test

Parameters:

`tipo` - es el tipo de respuesta que queremos dar al test

Class Protagonista

`java.lang.Object`
  `└ Protagonista`

All Implemented Interfaces:

java.lang.Runnable

```
public class Protagonista
extends java.lang.Object
implements java.lang.Runnable
```

Clase que define atributos y metodos que conforman al personaje protagonista de todo el juego. Dichos metodos implementarán el movimiento del personaje a lo largo de la escena, a igual que controlarán la ejecución del hilo en el que se ejecuta este objeto.

Since:

.JDK 1.4

See Also:

`Teclado`

Field Summary	
static int	**MUESTREO** Tiempo(ms) que duerme el hilo del objeto entre iteraciones consecutivas.
com.threed.jpct.Camera	**ojo** Elemento camera que representa la posicion de los ojos del personaje y la vista que tiene desde alli.
boolean	**vivo** Variable encargada de conservar vivo el hilo siempre y cuando se encuentre a true.

Constructor Summary
Protagonista(Mundo mundo, com.threed.jpct.SimpleVector posini, float alt, float rad, float caida, float movim, float giro) Constructor que define la forma, posición inicial, así como la velocidad de movimientos del protagonista.

Method Summary	
com.threed.jpct.SimpleVector	**getPosicion**() Método con el cual obtenemos la posición del protagonista
void	**mirarA**(float angl) Metodo que realiza el movimiento del protagonista que supone un giro con respecto a su eje hacia el angulo

	expresado como argumento.
vo:	**movAdelante**() Metodo que realiza el movimeinto del personaje hacia adelante, de modo que comprueba que no exista colisión.
void	**movAtras**() Metodo que realiza el movimeinto del personaje hacia atras, de modo que comprueba que no exista colisión.
void	**movCaer**() Metodo que realiza el movimeinto de caida del personaje, de modo que comprueba que no exista colisión.
void	**movDcha**() Metodo que realiza el movimiento del protagonista que supone un giro con respecto a su eje hacia la derecha.
void	**movIzda**() Metodo que realiza el movimiento del protagonista que supone un giro con respecto a su eje hacia la izquierda.
void	**run**() Metodo principal de la ejecución de un hilo y al que se llama una vez invocado el metodo start de Thread.
void	**setEdicion**(boolean edicion) Método por el cual indicamos que estamos en edicion
void	**setMaquinaEstados**(TransicionEstados transic) Selecciona la maquina de estados que caracteriza el Juego de la que poder consultar y tomar los cerrojos para conservar el correcto funcionamiento y el sincronismo entre instrucciones que lo requieran.
void	**setTeclado**(Teclado teclado) Asigna un controlador de teclado definido por la clase Teclado, que hara las funciones de mapeo del mismo para detectar los eventos producidos en él, y así poder actuar en consecuencia en el comportamiento y movimiento del protagonista.

Methods inherited from class java.lang.Object

clone, equals, finalize, getClass, hashCode, notify, notifyAll, toString, wait, wait, wait

Field Detail

MUESTREO

```
public static int MUESTREO
```

Tiempo(ms) que duerme el hilo del objeto entre iteraciones consecutivas. Tasa de muestreo con la que se realizan las operaciones de este objeto.

See Also:
run()

---

ojo

```
public com.threed.jpct.Camera ojo
```

Elemento camera que representa la posicion de los ojos del personaje y la vista que tiene desde alli. Propiamente dicho dicho elementento camera junto con las coordenadas que definen la elipse de colisión forman el cuerpo del protagonista.

---

vivo

```
public boolean vivo
```

Variable encargada de conservar vivo el hilo siempre y cuando se encuentre a true.

See Also:
run()

Constructor Detail

Protagonista

```
public Protagonista(Mundo mundo,
                    com.threed.jpct.SimpleVector posini,
                    float alt,
                    float rad,
                    float caida,
                    float movim,
                    float giro)
```

Constructor que define la forma, posición inicial, así como la velocidad de movimientos del protagonista. Además recibe un elemento mundo al que pertenecerá y del cual tomará sus características. El constructor crea una elipse de colisión con los parametros altura y radio que servirá para representar el cuerpo del protagonista y que se deberá adaptar a la escala del mundo que le rodea, de modo que este sea capaz de subir escaleras, ...

Parameters:

`mundo` - Objeto mundo donde se encuentra inscrito el protagonista, al que pertenece y del que toma sus características

`posini` - posicion inicial de ubicación del personaje

`alt` - Altura de la elipse que conforma el volumen del personaje

`rad` - Radio de grosor de la elipse que conforma el volumen del personaje

`caida` - Velocidad de caida del personaje

`movim` - Velocidad de movimiento del personaje

`giro` - Velocidad de giro del personaje

Method Detail

setEdicion

```
public void setEdicion(boolean edicion)
```

Método por el cual indicamos que estamos en edicion

Parameters:

`edicion,` - true si estamos editando, false en caso contrario

---

setMaquinaEstados

```
public void setMaquinaEstados(TransicionEstados transic)
```

Selecciónа la maquina de estados que caracteriza el Juego de la que poder consultar y tomar los cerrojos para conservar el correcto funcionamiento y el sincronismo entre instrucciones que lo requieran.
Parameters:
`transic` - Maquina de estados que asignaremos al protagonista
See Also:
`TransicionEstados`

---

movAdelante

`public void movAdelante()`

Metodo que realiza el movimeinto del personaje hacia adelante, de modo que comprueba que no exista colisión. De este modo si existiese colisión, unicamente se moveran tantas unidades como distancia halla hasta el objeto colisionado. Ciertas instrucciones se encuentran sincronizadas con la clase Representar de modo que unicamente se represente una vez se halla realizado el movimiento.
See Also:
`Representar.dibujarPantalla()`, `Representar.run()`

---

movAtras

`public void movAtras()`

Metodo que realiza el movimeinto del personaje hacia atras, de modo que comprueba que no exista colisión. De este modo si existiese colisión, unicamente se moveran tantas unidades como distancia halla hasta el objeto colisionado. Ciertas instrucciones se encuentran sincronizadas con la clase Representar de modo que unicamente se represente una vez se halla realizado el movimiento.
See Also:
`Representar.dibujarPantalla()`, `Representar.run()`

---

movIzda

`public void movIzda()`

Metodo que realiza el movimiento del protagonista que supone un giro con respecto a su eje hacia la izquierda. De este modo el metodo realiza un giro en la camara de forma que apunte a la dirección adecuada y actualiza la matriz de posicionamiento.

---

movDcha

`public void movDcha()`

Metodo que realiza el movimiento del protagonista que supone un giro con respecto a su eje hacia la derecha. De este modo el metodo realiza un giro en la camara de forma que apunte a la dirección adecuada y actualiza la matriz de posicionamiento.

---

mirarA

`public void mirarA(float angl)`

Metodo que realiza el movimiento del protagonista que supone un giro con respecto a su eje hacia el angulo expresado como argumento. De este modo el metodo realiza un giro en la camara de forma que apunte a la dirección adecuada y actualiza la matriz de posicionamiento.
Parameters:
`angl` - Angulo girado por la cámara

---

movCaer

`public void movCaer()`

Metodo que realiza el movimeinto de caida del personaje, de modo que comprueba que no exista colisión. De este modo si existiese colisión, unicamente se moveran tantas unidades como distancia halla hasta el objeto colisionado y para la caida. Ciertas instrucciones se encuentran sincronizadas con la clase Representar de modo que unicamente se represente una vez se halla realizado el movimiento.
See Also:
`Representar.dibujarPantalla()`, `Representar.run()`

---

setTeclado

`public void setTeclado(Teclado teclado)`

Asigna un controlador de teclado definido por la clase Teclado, que hara las funciones de mapeo del mismo para detectar los eventos producidos en él, y así poder actuar en consecuencia en el comportamiento y movimiento del protagonista.
Parameters:
`teclado` - Controlador del Teclado asignado

---

getPosicion

`public com.threed.jpct.SimpleVector getPosicion()`

Método con el cual obtenemos la posición del protagonista
Returns:
posición

---

run

`public void run()`

Metodo principal de la ejecución de un hilo y al que se llama una vez invocado el metodo start de Thread. El hilo seguirá vivo mientras la variable vivo se encuentre a true, en caso contrario se saldrá del bucle y terminará su ejecución. En el bucle se comprueba que el cerrojo oportuno de la maquina de estados este abierto y si así es produce un movimiento en el protagonista llamando al método privado moverProtagonista, encargado de gestiona si se ha producido algun evento en el teclado y si así fuera actua en consecuencia llamando al metodo oportuno de movimiento. En todas las llamadas se comprueba si el protagonista realiza un movimiento de caida o no Una vez terminada una vuelta al bucle se esperará el tiempo restante especificado en MUESTREO
Specified by:
`run` in interface `java.lang.Runnable`
See Also:
`TransicionEstados.TPgetState()`, `movAdelante()`, `movAtras()`, `movIzda()`, `movDcha()`, `MUESTREO`, `vivo`, `Teclado.getAdelante()`, `Teclado.getAtras()`, `Teclado.getDcha()`, `Teclado.getIzda()`, `Teclado.getSalir()`

Class Representar

```
java.lang.Object
  └ Representar
```

All Implemented Interfaces:
java.lang.Runnable

```
public class Representar
extends java.lang.Object
implements java.lang.Runnable
```

Clase que implementa metodos y atributos necesarios para la rederización y representación de las imagenes del juego. El objeto de esta clase se ejecutará en un hilo independiente y que realizará muestreos de la imagen actual cada cierto tiempo que vendrá dado por la variable MUESTREO.

Since:
.JDK 1.4

See Also:
MUESTREO, run(), MenuObjetos

Field Summary	
static int	**MUESTREO** Tiempo(ms) que duerme el hilo del objeto entre iteraciones consecutivas.
boolean	**vivo** Variable encargada de conservar vivo el hilo siempre y cuando se encuentre a true.

Constructor Summary
Representar(Mundo mundo, int ancho, int alto, java.awt.Graphics gFrame, java.awt.Frame frame, boolean software, TransicionEstados transic, java.lang.String directorio, Teclado tecla) Constructor del objeto representar que creará la ventana del juego y la dotará de las características adecuadas de configuración especificadas en el documento XML.

Method Summary	
void	**aumentar**() Metodo por el cual podemos aumentar un objeto 3D que queremos colocar en el escenario
void	**avanzar**() Metodo por los cuales movemos un objeto 3D en el escenario
void	**bajar**() Método con el que bajamos un objeto 3D por el eje Y
void	**colocar**() Método con el que colocamos un objeto 3D en el escenario
void	**crearDialogo**() Método por el que mostramos el dialogo por pantalla
void	**crearMenuSonido**() Método por el que se mostrará la ventana para que podamos decir si queremos sonido en la aplicación o no
void	**decrementar**() Metodo por el cual decrementamos la escala de un objeto 3D que queremos colocar en el escenario del juego
void	**derecha**() Método con el que movemos un objeto 3D a la derecha del eje X
void	**dibujarMenu**() Metodo que dibuja un cuadro de dialogo en la escena
java.lang.String	**getFichero**() Método por el que sabemos el nombre del fichero
Teclado	**getTeclado**() Método con el que obtenemos el objeto Teclado
void	**girar**() Metodo con el que giramos un objeto 3D sobre el eje Y
void	**girarX**() Metodo con el que giramos un objeto 3D sobre el eje X
void	**girarZ**() Metodo con el que giramos un objeto 3D sobre el eje Z
void	**guardar**(java.lang.String archivo, com.threed.jpct.SimpleVector pos, int id) Método con el que podemos guardar una partida
void	**izquierda**() Método con el que movemos un objeto 3D a la izquierda del X
boolean	**probar**() Con el que sabemos si después de editar un guión de juego queremos probar/jugar con lo que se acaba de editar
void	**retroceder**() Metodo con el que retrocedemos sobre el eje z

void	**run**() Metodo principal de la ejecución de un hilo y al que se llama una vez invocado el metodo start de Thread.
void	**setDialogo**(Dialogo dialogo) Método con el que modificamos el objeto Dialogo
void	**setEdicion**(boolean editar) Método con el que indicamos si estamos editando
void	**setFases**(java.util.Vector fases) Método por el cual modificamos las fases iniciales del objeto menu para la edición.
void	**setFichero**(java.lang.String fichero) Método que modifica el nombre del fichero.
void	**setInicio**(java.lang.String archivo) Método por lo que le indicamos al objeto menu el archivo con el cual arrancamos la aplicación, para que así sepa de dónde tiene que ir cargando los objetos 3D y archivos docentes
void	**setProbar**(boolean probar) Método con el que modificamos la variable probar
void	**setProtagonista**(com.threed.jpct.SimpleVector prota) Método con el que indicamos la posicion del protagonista para generar el guion editado
void	**setTeclado**(Teclado tecla) Método con el que modificamos el objeto teclado
void	**subir**() Método con el que subimos un objeto 3D por el eje Y
void	**verMenu**() Metodo con el que hacemos visible el menu de sonido

Methods inherited from class java.lang.Object
clone, equals, finalize, getClass, hashCode, notify, notifyAll, toString, wait, wait, wait

Field Detail

MUESTREO

```
public static int MUESTREO
```

Tiempo(ms) que duerme el hilo del objeto entre iteraciones consecutivas. Tasa de muestreo con la que se realizan las operaciones de este objeto.

See Also:

run()

---

vivo

```
public boolean vivo
```

Variable encargada de conservar vivo el hilo siempre y cuando se encuentre a true.

See Also:

Protagonista.run()

Constructor Detail

Representar

```
public Representar(Mundo mundo,
                   int ancho,
                   int alto,
                   java.awt.Graphics gFrame,
                   java.awt.Frame frame,
                   boolean software,
                   TransicionEstados transic,
                   java.lang.String directorio,
                   Teclado tecla)
```

Constructor del objeto representar que creará la ventana del juego y la dotará de las características adecuadas de configuración especificadas en el documento XML. Este constructor asignara un objeto mundo del que tomará las caracteristicas la representacion y rederización, así como el tamaño de la pantalla, modo de renderización y maquina de estados que caracteriza el juego de la que poder tomar los cerrojos pertienetes conservando así el correcto funcionamiento y el sincronismo entre instrucciones que lo requieran.

Parameters:

`mundo` - Objeto mundo del que se toman caracteristicas de configuración.
`ancho` - Ancho de la ventana de representación
`alto` - Alto de la ventana de representacion
`gFrame` - Elemento controlador de los gráficos
`frame` - Ventana en la que representar el juego
`software` - Boolean que indica si existe renderización por software (true) o por hardware (false)
`transic` - Maquina de estados del juego
`directorio` - es el directorio donde esta el guion
`tecla` - es el que maneja el teclado

Method Detail

setDialogo

```
public void setDialogo(Dialogo dialogo)
```

Método con el que modificamos el objeto Dialogo

Parameters:

`dialogo` - nuevo

---

setInicio

```
public void setInicio(java.lang.String archivo)
```

Método por lo que le indicamos al objeto menu el archivo con el cual arrancamos la aplicación, para que así sepa de dónde tiene que ir cargando los objetos 3D y archivos docentes

---

guardar

```
public void guardar(java.lang.String archivo,
                    com.threed.jpct.SimpleVector pos,
                    int id)
```

Método con el que podemos guardar una partida

Parameters:

`archivo` - es la ruta del archivo con el que estamos jugando

`pos` - es la posicion del protagonista

`id` - es el id de la fase actual

---

dibujarMenu

```
public void dibujarMenu()
```

Metodo que dibuja un cuadro de dialogo en la escena

---

crearDialogo

```
public void crearDialogo()
```

Método por el que mostramos el dialogo por pantalla

---

crearMenuSonido

```
public void crearMenuSonido()
```

Método por el que se mostrará la ventana para que podamos decir si queremos sonido en la aplicación o no

---

aumentar

```
public void aumentar()
```

Metodo por el cual podemos aumentar un objeto 3D que queremos colocar en el escenario

---

decrementar

```
public void decrementar()
```

Metodo por el cual decrementamos la escala de un objeto 3D que queremos colocar en el escenario del juego

---

avanzar

```
public void avanzar()
```

Metodo por los cuales movemos un objeto 3D en el escenario

---

retroceder

```
public void retroceder()
```

Metodo con el que retrocedemos sobre el eje z

---

subir

```
public void subir()
```

Método con el que subimos un objeto 3D por el eje Y

---

bajar

```
public void bajar()
```

Método con el que bajamos un objeto 3D por el eje Y

---

izquierda

```
public void izquierda()
```

Método con el que movemos un objeto 3D a la izquierda del X

---

derecha

```
public void derecha()
```

Método con el que movemos un objeto 3D a la derecha del eje X

---

girar

```
public void girar()
```

Metodo con el que giramos un objeto 3D sobre el eje Y

---

girarZ

```
public void girarZ()
```

Metodo con el que giramos un objeto 3D sobre el eje Z

---

girarX

```
public void girarX()
```

Metodo con el que giramos un objeto 3D sobre el eje X

---

colocar

```
public void colocar()
```

Método con el que colocamos un objeto 3D en el escenario

---

setTeclado

`public void setTeclado(Teclado tecla)`

Método con el que modificamos el objeto teclado

Parameters:

`nuevo` - teclado

---

getTeclado

`public Teclado getTeclado()`

Método con el que obtenemos el objeto Teclado

Returns:

teclado

---

setEdicion

`public void setEdicion(boolean editar)`

Método con el que indicamos si estamos editando

Parameters:

`true` - si editamos, false en caso contrario

---

setProtagonista

`public void setProtagonista(com.threed.jpct.SimpleVector prota)`

Método con el que indicamos la posicion del protagonista para generar el guion editado

Parameters:

`prota` - posicion de la camara

---

setFases

`public void setFases(java.util.Vector fases)`

Método por el cual modificamos las fases iniciales del objeto menu para la edición. Utilizado sólo en caso de que el guión con el que partimos para editar ya tenga creada alguna fase anterior.

Parameters:

`fases` - es el vector que contiene las fases editadas anteriormente con las que debemos generar el nuevo guión.

---

setFichero

`public void setFichero(java.lang.String fichero)`

Método que modifica el nombre del fichero. Utilizado inicialmente para que en el modo edición la clase MenuObjeto sepa en qué directorio debe buscar las cosas

---

probar

`public boolean probar()`

Con el que sabemos si después de editar un guión de juego queremos probar/jugar con lo que se acaba de editar

---

setProbar

`public void setProbar(boolean probar)`

Método con el que modificamos la variable probar

Parameters:

`probar` - boolean

---

getFichero

`public java.lang.String getFichero()`

Método por el que sabemos el nombre del fichero

---

verMenu

`public void verMenu()`

Metodo con el que hacemos visible el menu de sonido

---

run

`public void run()`

Metodo principal de la ejecución de un hilo y al que se llama una vez invocado el metodo start de Thread. El hilo seguirá vivo mientras la variable vivo se encuentre a true, en caso contrario se saldrá del bucle y terminará su ejecución. En el bucle se comprueba que el cerrojo oportuno de la maquina de estados este abierto y si así es se produce una representación de la escena actual llamando al método privado dibujar, encargado de dibujar el frame con la imagen rederizada actual del juego. Para ello borra el bufer, renderiza la imagen y la mete en el buffer para posteriormente dibujarla segun el modo apropiado. Una vez terminada una vuelta al bucle se esperará el tiempo restante especificado en MUESTREO. En el caso de recibir una orden de cierre del hilo, se terminará el bucle y se llamará a la instrucción privada cerrarBuffer que libera los recursos utilizados en la representación.

Specified by:

`run` in interface `java.lang.Runnable`

See Also:

`MUESTREO`, `vivo`

Class Tema

`java.lang.Object`
  └ **Tema**

`public class Tema`

extends java.lang.Object

Clase con la que podemos crear/modificar un tema docente para posteriormente generar el correspondiente fichero

---

Constructor Summary
Tema(java.lang.String contenido) Constructor de la clase
Tema(java.lang.String titulo, java.lang.String contenido) Constructor de la clase

Method Summary	
java.lang.String	**getContenido**() Metodo por el cual obtenemos el contenido del tema
java.lang.String	**getTitulo**() Metodo por el cual obtenemos el titulo del tema
void	**setContenido**(java.lang.String contenido) Metodo por el cual modificamos el contenido del tema
void	**setTitulo**(java.lang.String titulo) Metodo por el cual modificamos el titulo del tema

Methods inherited from class java.lang.Object
clone, equals, finalize, getClass, hashCode, notify, notifyAll, toString, wait, wait, wait

Constructor Detail

Tema

public **Tema**(java.lang.String contenido)

Constructor de la clase

Parameters:

contenido - es el contenido del tema

Tema

public **Tema**(java.lang.String titulo,
java.lang.String contenido)

Constructor de la clase

Parameters:

titulo - es el titulo del tema

contenido - es el contenido del tema

Method Detail

setTitulo

public void **setTitulo**(java.lang.String titulo)

Metodo por el cual modificamos el titulo del tema

Parameters:

titulo - es el nuevo titulo que queremos dar al tema

---

setContenido

public void **setContenido**(java.lang.String contenido)

Metodo por el cual modificamos el contenido del tema

Parameters:

contenido - es el nuevo contenido que queremos dar al tema

---

getTitulo

public java.lang.String **getTitulo**()

Metodo por el cual obtenemos el titulo del tema

Returns:

titulo del tema

---

getContenido

public java.lang.String **getContenido**()

Metodo por el cual obtenemos el contenido del tema

Returns:

contenido del tema

Class Temario

java.lang.Object
└ org.jdom.Content

└ org.jdom.DocType
  └ **Temario**

All Implemented Interfaces:
java.io.Serializable, java.lang.Cloneable

```
public class Temario
extends org.jdom.DocType
```

Clase que se emplea para crear el documento xml referente a los temas docentes

Since:
.JDK 1.4

See Also:
Docencia, Serialized Form

Field Summary

Fields inherited from class org.jdom.DocType

`elementName, internalSubset, publicID, systemID`

Fields inherited from class org.jdom.Content

`parent`

Constructor Summary

Temario() Constructor por defecto de la clase
Temario(java.lang.String fichero) Constructor de la clase
Temario(java.util.Vector temas) Constructor de la clase

Method Summary

void	**construirTemario**(org.jdom.Element temario) Metodo por el cual tras pasar un fichero en el constructor podemos crear los objetos temas
org.jdom.DocType	**crearDocType**() Metodo por el cual le decimos al documento xml cual es el dtd que tiene que seguir
void	**crearFichero**() Metodo por el cual creamos el fichero xml con los temas utilizados por la aplicacion
java.lang.String	**getFichero**() Metodo por el que obtenemos el nombre del fichero
java.util.Vector	**getTemas**() Metodo por el cual obtenemos todos los temas de la aplicacion
void	**setNombre**(java.lang.String nombre) Metodo por el que modificamos el nombre del fichero
void	**setTemas**(java.util.Vector temas) Metodo por el cual modificamos los temas

Methods inherited from class org.jdom.DocType

`getElementName, getInternalSubset, getPublicID, getSystemID, getValue, setElementName, setInternalSubset, setPublicID, setSystemID, toString`

Methods inherited from class org.jdom.Content

`clone, detach, equals, getDocument, getParent, getParentElement, hashCode, setParent`

Methods inherited from class java.lang.Object

`finalize, getClass, notify, notifyAll, wait, wait, wait`

Constructor Detail

Temario

```
public Temario(java.util.Vector temas)
```

Constructor de la clase

Parameters:
`temas` - es el vector que contiene los temas de la aplicacion

Temario

```
public Temario()
```

Constructor por defecto de la clase

Temario

`public Temario(java.lang.String fichero)`

Constructor de la clase

Parameters:

`fichero` - es el nombre del fichero donde puede haber temas docentes que se utilicen por la aplicación

Method Detail

crearDocType

`public org.jdom.DocType crearDocType()`

Metodo por el cual le decimos al documento xml cual es el dtd que tiene que seguir

Returns:

doctype del dtd

setNombre

`public void setNombre(java.lang.String nombre)`

Metodo por el que modificamos el nombre del fichero

Parameters:

`nombre` - nuevo

getFichero

`public java.lang.String getFichero()`

Metodo por el que obtenemos el nombre del fichero

Returns:

nombre

setTemas

`public void setTemas(java.util.Vector temas)`

Metodo por el cual modificamos los temas

Parameters:

`temas` - son los nuevos temas del temario

getTemas

`public java.util.Vector getTemas()`

Metodo por el cual obtenemos todos los temas de la aplicacion

Returns:

temas

construirTemario

`public void construirTemario(org.jdom.Element temario)`

Metodo por el cual tras pasar un fichero en el constructor podemos crear los objetos temas

Parameters:

`temario` - es el elemento raiz del fichero xml que contiene los temas docentes

crearFichero

`public void crearFichero()`

Metodo por el cual creamos el fichero xml con los temas utilizados por la aplicacion

Class Test

```
java.lang.Object
  └ Test
```

`public class Test extends java.lang.Object`

La clase Test es la clase que representa un test docente

Since:

.JDK 1.4

See Also:

`NuevoTest`, `Cuestionario`

Constructor Summary

`Test()` Constructor de la clase
`Test(java.util.Vector preguntas)` Constructor de la clase

Method Summary

java.lang.String	**getNombre**() Método por el que obtenemos el nombre del fichero del test
int	**getNumPregunta**() Metodo por el cual sabemos el numero de preguntas que tiene el test
java.util.Vector	**getPreguntas**() Metodo con el cual obtenemos las preguntas (y respuestas) del test
void	**setNombre**(java.lang.String nombre) Método por el que indicamos cómo se llamará el fichero test
void	**setPreguntas**(java.util.Vector preguntas) Metodo por el cual modificamos las preguntas que contiene el test

Methods inherited from class java.lang.Object

clone, equals, finalize, getClass, hashCode, notify, notifyAll, toString, wait, wait, wait

Constructor Detail

Test

public **Test**()
Constructor de la clase

Test

public **Test**(java.util.Vector preguntas)
Constructor de la clase
Parameters:
preguntas - es el vector que contienes los elementos preguntas

Method Detail

setPreguntas

public void **setPreguntas**(java.util.Vector preguntas)
Metodo por el cual modificamos las preguntas que contiene el test
Parameters:
preguntas - son las preguntas que tendra el test

---

setNombre

public void **setNombre**(java.lang.String nombre)
Método por el que indicamos cómo se llamará el fichero test
Parameters:
nombre - String del nombre del fichero

---

getNombre

public java.lang.String **getNombre**()
Método por el que obtenemos el nombre del fichero del test
Returns:
String nombre fichero

---

getPreguntas

public java.util.Vector **getPreguntas**()
Metodo con el cual obtenemos las preguntas (y respuestas) del test
Returns:
preguntas del test

---

getNumPregunta

public int **getNumPregunta**()
Metodo por el cual sabemos el numero de preguntas que tiene el test
Returns:
numero de preguntas del test

Class Ventana

java.lang.Object
  └ GUIComponent
    └ **Ventana**

---

public class **Ventana**
extends GUIComponent

La clase ventana, puede contener otros objetos (como botones, campo de texto...). La imagen de la ventana se define por una textura.

Field Summary

Fields inherited from class GUIComponent

caracter, visible

Constructor Summary

Ventana(com.threed.jpct.Texture ventana, int x, int y) Constructor de la clase	

Method Summary

void	**draw**(com.threed.jpct.FrameBuffer buffer) Metodo por el que dibujamos la ventana
boolean	**evaluateInput**(MouseMapper mouse, com.threed.jpct.util.KeyMapper keyMapper) Metodo que evalua si se ha producido un evento en la ventana
void	**setTextura**(com.threed.jpct.Texture ventana) Metodo con el cambiamos la textura definida como ventana

Methods inherited from class GUIComponent

add, getParent, getParentX, getParentY, getX, getY, isVisible, remove, setCaracter, setVisible, setX, setXFigura, setY, setYFigura

Methods inherited from class java.lang.Object

clone, equals, finalize, getClass, hashCode, notify, notifyAll, toString, wait, wait, wait

Constructor Detail

Ventana

```
public Ventana(com.threed.jpct.Texture ventana,
               int x,
               int y)
```

Constructor de la clase

Parameters:

ventana - es la textura que define la ventana.

x - posición x inicial de la ventana (esquina superior izquierda).

y - posición y inicial de la ventana (esquina superior izquierda).

Method Detail

setTextura

```
public void setTextura(com.threed.jpct.Texture ventana)
```

Metodo con el cambiamos la textura definida como ventana

Parameters:

ventana - la nueva textura

---

evaluateInput

```
public boolean evaluateInput(MouseMapper mouse,
                             com.threed.jpct.util.KeyMapper keyMapper)
```

Metodo que evalua si se ha producido un evento en la ventana

Overrides:

evaluateInput in class GUIComponent

Parameters:

mouse - representa el raton (si hemos pinchado)

keyMapper - representa el teclado (si hemos presionado una tecla)

Returns:

boolean true si ocurrio un evento, false en caso contrario

---

draw

```
public void draw(com.threed.jpct.FrameBuffer buffer)
```

Metodo por el que dibujamos la ventana

Overrides:

draw in class GUIComponent

Parameters:

buffer - FrameBuffer donde dibujamos

BIBLIOGRAFÍA

Organizada por orden alfabético de autor, según tipo de documento:

Libros

[1] Deitel y Deitel: *Como programar en Java*. Editorial Prentice Hall. 1998.

[2] Naughton, P. & Schildt, H.: *Java. Manual de Referencia.* Editorial: McGraw Hill. 1997.

[3] Prensky, M.: *Digital Game-Based Learning*. Editorial McGraw-Hill. Diciembre, 2000.

Artículos

[4] Catá, Jordi: *Comparativa de motores gráficos para videojuegos.* Mayo, 2002, disponible en: http://ima.udg.es/~amendez/TIC2001/docs/Engines.pdf [Consulta: 29 abril 2009].

[5] Gómez-Martín Marco A., Gómez-Martín Pedro P. y González-Calero Pedro A.: *Aprendizaje basado en juegos.* Disponible en: http://www.icono14.net/revista/num4/marco_antonio.doc [Consulta: 29 abril 2009].

[6] Grupo de informática gráfica: *Motores de juego.* Disponible en: http://ima.udg.es/~amendez/TIC2001/docs/EnginesUJI.pdf [Consulta: 29 abril 2009].

[7] Talón Argente Rubén: *Motores Gráficos.*

Direcciones web

[8] *3D GameStudio*. Disponible en: http://www.3dgamestudio.com/ [Consulta: 29 abril 2009].

[9] ELEARNINGEUROPA - The *benefits of the game-based learning*. Disponible en: http://www.elearningeuropa.info/index.php [Consulta: 29 abril 2009].

[10] *JavaHispano*. Disponible en: http://www.javahispano.org [Consulta: 25 mayo 2009].

[11] *jPCT*. Disponible en: http://www.jpct.net [Consulta: 24 noviembre 2009].

[12] *Tutorial Blender*. Disponible en: http://www.wiki.blender.org/index.php/Doc:ES/Manual [Consulta: 25 mayo 2009]

[13] *Wikipedia*. Disponible en: http://es.wikipedia.org/ [Consulta: 24 noviembre 2009].

PFC

[14] Jesús Sáez Gómez Escalonilla: *Plataforma de juegos basada en ejecución de guiones.* Editorial UC3M, 2007